VISIT…

Çetin Kaya Koç
Editor

Cryptographic Engineering

Springer

Cryptographic Engineering

Çetin Kaya Koç
Editor

Cryptographic Engineering

Springer

Editor
Çetin Kaya Koç
City University of Istanbul
Tophane, Istanbul
Turkey
and
University of California Santa Barbara
Santa Barbara, CA
USA

ISBN: 978-0-387-71816-3 e-ISBN: 978-0-387-71817-0
DOI 10.1007/978-0-387-71817-0

Library of Congress Control Number: 2008935379

© Springer Science+Business Media, LLC 2009
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.
While the advice and information in this book are believed to be true and accurate at the date of going to press, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

springer.com

To all scientists and engineers whose ideas gave birth to modern cryptography, particularly, Claude Shannon, Whit Diffie, Martin Hellman, Ralph Merkle, Don Coppersmith, Ron Rivest, Adi Shamir, Len Adleman, and Neal Koblitz.

Preface

Cryptography is an ancient art. Chinese, Roman, and Arab cultures often used ciphers to protect military and state communications or secret society documents. Cryptographic engineering, on the other hand, is a relatively new subject. A cryptographic engineer designs, implements, tests, validates, and sometimes reverse-engineers or attempts to break cryptographic systems. The designers of Enigma, an electromechanical cipher machine, were cryptographic engineers; so was Alan Turing who contributed to its cryptanalysis. In our view, anyone who designs and builds electromechanical, electronic, or quantum-mechanical systems in order to encrypt, decrypt, sign or authenticate data is a cryptographic engineer. However, in this book we have narrowed our definition to only electronic systems, specifically, hardware and software systems.

Cryptographic engineering is a complicated, multidisciplinary field. It encompasses mathematics (algebra, finite groups, rings, and fields), electrical engineering (hardware design, ASIC, FPGAs) and computer science (algorithms, complexity theory, software design, embedded systems). It is rather difficult to be a master of all subjects; one usually has to be content with being a master of one. In order to practice state-of-the-art cryptographic design, mathematicians, computer scientists, and electrical engineers need to collaborate.

This book was born out of the class notes of the lecturers who have been meeting since 2002 in Lausanne, Switzerland, at the campus of EPFL, to teach a one-week course to graduate students, faculty, and researchers from academia, and engineers from industry. In order to create this book, I compiled the lecture notes together, wrote some of the material, and also invited other prominent researchers to contribute. This book is intended to constitute a first step towards becoming a cryptographic engineer. We hope that it will successfully serve its purpose.

Istanbul & Santa Barbara *Çetin Kaya Koç*

Contents

1 About Cryptographic Engineering 1
Çetin Kaya Koç
1.1 Introduction 1
1.2 Chapter Contents 2
1.3 Exercises and Projects 4

2 Random Number Generators for Cryptographic Applications 5
Werner Schindler
2.1 Introduction 5
2.2 General Requirements 6
2.3 Classification 7
2.4 Deterministic Random Number Generators (DRNGs) 7
2.4.1 Pure DRNGs 8
2.4.2 Hybrid DRNGs 11
2.4.3 A Word of Warning 13
2.5 Physical True Random Number Generators (PTRNGs) 14
2.5.1 The Generic Design 14
2.5.2 Entropy and Guesswork 16
2.6 Non-physical True Random Number Generators (NPTRNGs): Basic Properties 18
2.7 Standards and Evaluation Guidances 20
2.8 Exercises 20
2.9 Projects 21
References 21

3 Evaluation Criteria for Physical Random Number Generators 25
Werner Schindler
3.1 Introduction 25
3.2 Generic Design 26
3.3 Evaluation Criteria for the Principle Design 27
3.4 The Stochastic Model 29

3.5 Algorithmic Postprocessing . . . 37
3.6 Online Test, Tot Test, and Self Test . . . 41
3.6.1 Online Tests . . . 42
3.7 Alternative Security Philosophies . . . 49
3.8 Side-channel Attacks and Fault Attacks . . . 50
3.9 Exercises . . . 51
3.10 Projects . . . 51
References . . . 52

4 True Random Number Generators for Cryptography . . . 55
Berk Sunar
4.1 Introduction . . . 55
4.2 TRNG Building Blocks . . . 56
4.3 Desirable Features . . . 57
4.4 Survey of TRNG Designs . . . 57
4.4.1 Baggini and Bucci . . . 58
4.4.2 The Intel TRNG Design . . . 58
4.4.3 The Tkacik TRNG Design . . . 59
4.4.4 The Epstein et al. TRNG Design . . . 60
4.4.5 The Fischer–Drutarovský Design . . . 61
4.4.6 The Golić FIGARO Design . . . 62
4.4.7 The Kohlbrenner–Gaj Design . . . 63
4.4.8 The Bucci–Luzzi Testable TRNG Design Framework . . . 64
4.4.9 The Rings Design . . . 65
4.4.10 The PUF–RNG Design . . . 66
4.4.11 The Yoo et al. Design . . . 67
4.4.12 The Dichtl and Golić RNG Design . . . 67
4.5 Postprocessing Techniques . . . 68
4.6 Exercises . . . 70
References . . . 71

5 Fast Finite Field Multiplication . . . 75
Serdar Süer Erdem, Tuğrul Yanık, and Çetin Kaya Koç
5.1 Introduction . . . 75
5.2 Finite Fields . . . 76
5.3 Multiplication in Prime Fields . . . 77
5.3.1 Integer Multiplication . . . 78
5.3.2 Integer Squaring . . . 80
5.3.3 Integer Modular Reduction . . . 80
5.4 Multiplication in Binary Extension Fields . . . 87
5.4.1 Polynomial Multiplication over $\mathbb{F}_2$ . . . 88
5.4.2 Polynomial Squaring over $\mathbb{F}_2$ . . . 90
5.4.3 Polynomial Modular Reduction over $\mathbb{F}_2$ . . . 90
5.5 Multiplication in General Extension Fields . . . 96
5.5.1 Field Multiplication in OEF . . . 97
5.5.2 Coefficient Multiplication and Reductions . . . 98

5.6 Karatsuba–Ofman Algorithm 99
5.6.1 Complexity 100
5.6.2 Number of Scalar Multiplications 100
5.7 Exercises 102
5.8 Projects 103
References 103

6 Efficient Unified Arithmetic for Hardware Cryptography 105
Erkay Savaş and Çetin Kaya Koç
6.1 Introduction 105
6.2 Fundamentals of Extension Fields 106
6.3 Addition and Subtraction 107
6.4 Multiplication 110
6.4.1 Montgomery Multiplication Algorithm 110
6.4.2 Dual-Radix Multiplier 116
6.4.3 Support for Ternary Extension Fields, GF(3^n) 118
6.5 Inversion 119
6.5.1 Montgomery Inversion for $GF(p)$ and $GF(2^n)$ 119
6.6 Conclusions 122
6.7 Exercises 122
6.8 Projects 123
References 123

7 Spectral Modular Arithmetic for Cryptography 125
Gökay Saldamlı and Çetin Kaya Koç
7.1 Introduction 125
7.2 Notation and Background 126
7.2.1 Evaluation Polynomials 126
7.2.2 Discrete Fourier Transform (DFT) 129
7.2.3 Properties of DFT: Time–frequency dictionary 131
7.3 Spectral Modular Arithmetic 135
7.3.1 Time Simulations and Spectral Algorithms 135
7.3.2 Modular Reduction 136
7.3.3 Spectral Modular Reduction 137
7.3.4 Time Simulation of Spectral Modular Reduction 139
7.3.5 Spectral Modular Reduction in a Finite Ring Spectrum 141
7.3.6 Spectral Modular Multiplication (SMM) 143
7.3.7 Spectral Modular Exponentiation 145
7.3.8 Illustrative Example 149
7.4 Applications to Cryptography 153
7.4.1 Mersenne and Fermat rings 154
7.4.2 Pseudo Number Transforms 155
7.4.3 Parameter Selection for RSA 156
7.4.4 Parameter Selection for ECC over Prime Fields 157
7.5 Spectral Extension Field Arithmetic 158
7.5.1 Binary Extension Fields 158

7.5.2 Midsize Characteristic Extension Fields . . . 161
7.5.3 Parameter Selection for ECC over Extension Fields . . . 164
7.6 Notes . . . 165
7.7 Exercises . . . 166
7.8 Projects . . . 167
References . . . 168

8 Elliptic and Hyperelliptic Curve Cryptography . . . 171
Nigel Boston and Matthew Darnall
8.1 Introduction . . . 171
8.2 Diffie – Hellman Key Exchange . . . 172
8.3 Introduction to Elliptic and Hyperelliptic Curves . . . 172
8.4 The Jacobian of a Curve . . . 173
8.4.1 The Principal Subgroup and $Jac(C)$ . . . 174
8.5 Computing on $Jac(C)$ . . . 174
8.6 Group Law for Elliptic Curves . . . 176
8.7 Techniques for Computations in Hyperelliptic Curves . . . 178
8.7.1 Explicit Formulae . . . 178
8.7.2 Projective Coordinates . . . 178
8.7.3 Other Optimization Techniques . . . 179
8.8 Counting Points on $Jac(C)$ . . . 179
8.9 Attacks . . . 181
8.9.1 Baby-Step Giant-Step Attack . . . 181
8.9.2 Pollard Rho and Lambda Attacks . . . 181
8.9.3 Pohlig–Hellman Attack . . . 182
8.9.4 Menezes–Okamoto–Vanstone Attack . . . 182
8.9.5 Semaev, Satoh-Araki, Smart Attack . . . 183
8.9.6 Attacks employing Weil descent . . . 183
8.10 Good Curves . . . 184
8.11 Exercises . . . 184
8.12 Projects . . . 185
References . . . 185

9 Instruction Set Extensions for Cryptographic Applications . . . 191
Sandro Bartolini, Roberto Giorgi, and Enrico Martinelli
9.1 Introduction . . . 191
9.1.1 Instruction Set Architecture . . . 191
9.2 Applications and Benchmarks . . . 194
9.2.1 Benchmarks . . . 195
9.2.2 Potential Performance . . . 195
9.3 ISE for Cryptographic Applications . . . 196
9.3.1 Instructions for Information Confusion and Diffusion . . . 196
9.3.2 ISE for AES . . . 203
9.3.3 ISE for ECC applications . . . 212
9.4 Exercises . . . 227
9.5 Projects . . . 228
References . . . 229

10 FPGA and ASIC Implementations of AES 235
Kris Gaj and Pawel Chodowiec
10.1 Introduction 235
10.2 AES Cipher Description 236
10.2.1 Basic Features 236
10.2.2 Round Operations 237
10.2.3 Iterative Structure 242
10.2.4 Key Scheduling 243
10.3 FPGA and ASIC Technologies 247
10.4 Parameters of Hardware Implementations 250
10.4.1 Throughput and Latency 250
10.4.2 Area 250
10.5 Hardware Architectures of Symmetric Block Ciphers 251
10.5.1 Hardware Architectures vs. Block Cipher Modes of Operation 251
10.5.2 Basic Iterative Architecture 252
10.5.3 Loop Unrolling 253
10.5.4 Pipelining 254
10.5.5 Limits on the Maximum Clock Frequency of Pipelined Architectures 258
10.5.6 Compact Architectures with Resource Sharing 260
10.6 Implementation of Basic Operations of AES in Hardware 261
10.6.1 SubBytes and InvSubBytes 261
10.6.2 MixColumns and InvMixColumns 270
10.7 Hardware Architectures of a Single Round of AES 274
10.7.1 S-Box-Based Architecture 274
10.7.2 T-Box-Based Architecture 276
10.7.3 Compact Architectures 282
10.8 Implementation of Key Scheduling 286
10.9 Optimum Choice of a Hardware Architecture for AES 286
10.10 Exercises 289
10.11 Projects 290
References 291

11 Secure and Efficient Implementation of Symmetric Encryption Schemes using FPGAs 295
François-Xavier Standaert
11.1 Introduction 295
11.2 Efficient FPGA Implementations 297
11.2.1 Exploiting the Slice Structure 297
11.2.2 Exploiting Embedded Blocks 300
11.2.3 Exploiting Further Features 302
11.2.4 Combining the Tricks: The Flexibility *Versus* Efficiency Tradeoff 303
11.3 Fair Evaluation of a Cryptographic FPGA Design 303

11.3.1 Design Goals . . . 304
11.3.2 Performance Evaluation . . . 304
11.4 Security of FPGAs Against Side-Channel Attacks . . . 305
11.4.1 Applicability of the Attack and FPGA Properties . . . 306
11.4.2 Countermeasures . . . 310
11.4.3 Measuring Side-Channel Resistance . . . 311
11.5 Other Security Issues . . . 312
11.5.1 Fault Attacks . . . 312
11.5.2 Bitstream Security . . . 312
11.6 Conclusions and Open Questions . . . 315
11.7 Exercises . . . 315
11.8 Projects . . . 317
References . . . 318

12 Block Cipher Modes of Operation from a Hardware Implementation Perspective . . . 321
Debrup Chakraborty and Francisco Rodríguez-Henríquez
12.1 Introduction . . . 321
12.2 Block Ciphers . . . 326
12.3 Introduction to AES . . . 327
12.3.1 Byte Substitution (BS) Step . . . 328
12.3.2 Shift Rows (SR) Step . . . 328
12.3.3 Mix Columns (MC) Step . . . 328
12.3.4 Add Round Key (ARK) Step . . . 329
12.3.5 Key Scheduling Algorithm . . . 329
12.4 A Background in Binary Extension Finite Fields . . . 330
12.4.1 Rings . . . 330
12.4.2 Fields . . . 331
12.4.3 Finite Fields . . . 331
12.4.4 Binary Finite Field Arithmetic . . . 331
12.5 Traditional Modes of Operations . . . 332
12.5.1 Electronic Code Book Mode . . . 333
12.5.2 Cipher Block Chaining Mode . . . 333
12.5.3 Cipher Feedback Mode . . . 334
12.5.4 Output Feedback Mode . . . 334
12.5.5 Counter Mode . . . 335
12.6 Security Requirements for Modes of Operations . . . 336
12.6.1 The Adversary . . . 336
12.6.2 Privacy Only Modes . . . 337
12.6.3 Authenticated Encryption . . . 338
12.6.4 Disk Encryption Schemes . . . 339
12.6.5 Security Proofs . . . 341
12.7 Some Modern Modes . . . 341
12.7.1 The Offset Codebook Mode . . . 343
12.7.2 ECB-Mask-ECB Mode . . . 344

12.8 The CCM Mode: A Case Study 347
12.8.1 The CCM Mode 347
12.8.2 AES Encryptor Core Implementation 350
12.8.3 Hardware Implementation of the CCM Mode 353
12.8.4 Experimental Results and Comparison 357
12.9 Conclusions 358
12.10 Exercises 359
12.11 Projects 359
References 360

13 Basics of Side-Channel Analysis 365
Marc Joye
13.1 Introduction 365
13.2 Timing Analysis 365
13.2.1 Attack on a Password Verification 366
13.2.2 Attack on an RSA Signature Scheme 367
13.3 Simple Power Analysis 368
13.3.1 Reverse-Engineering of an Algorithm 369
13.3.2 Attack on a Private RSA Exponentiation 370
13.3.3 Attack on a DES Key Schedule 371
13.4 Differential Power Analysis 373
13.4.1 Bit Tracing 373
13.4.2 Attack on an AES Implementation 374
13.4.3 Attack on an RSA Signature Scheme (2) 376
13.5 Countermeasures 376
13.6 Exercises 377
13.7 Projects 378
References 379

14 Improved Techniques for Side-Channel Analysis 381
Pankaj Rohatgi
14.1 Introduction 381
14.2 CMOS Devices: Side-Channel Leakage Perspective 382
14.2.1 Intentional Current Flows 382
14.2.2 Leakage Current Flows 383
14.2.3 Information Leakage in Power and EM Side-Channels . . 383
14.3 Characterizing Side-Channel Leakage Using Maximum Likelihood 385
14.3.1 Adversarial Model 385
14.3.2 Maximum Likelihood and Best Attack Strategy 385
14.3.3 Gaussian Assumption 386
14.4 Template Attacks 387
14.4.1 Classical Template Attacks: The Case of RC4 389
14.4.2 Single-Bit Templates and Applications 393
14.5 Improved DPA/DEMA Metric 395
14.5.1 Improving DPA 395

14.6 Multi-Channel Attacks . . . 397
14.6.1 Multiple Channel Selection . . . 397
14.6.2 Multi-Channel Template Attacks . . . 399
14.6.3 Multi-Channel DPA . . . 400
14.7 Toward Information Leakage Assessment . . . 401
14.7.1 Practical Considerations . . . 402
14.8 Projects . . . 403
References . . . 405

15 Electromagnetic Attacks and Countermeasures . . . 407
Pankaj Rohatgi
15.1 Introduction and History . . . 407
15.2 EM Emanations Background . . . 409
15.2.1 Types of EM Emanations . . . 409
15.2.2 EM Propagation . . . 410
15.3 EM Capturing Equipment . . . 413
15.4 EM Leakage Examples . . . 415
15.4.1 Examples: Amplitude Modulation . . . 415
15.4.2 Examples: Angle Modulation . . . 422
15.5 Multiplicity of EM Channels and Comparison with Power Channel . . . 424
15.6 Using EM to Bypass Power Analysis Countermeasures . . . 427
15.7 Quantifying EM Exposure . . . 427
15.8 Countermeasures . . . 428
15.9 Projects . . . 429
References . . . 430

16 Leakage from Montgomery Multiplication . . . 431
Colin D. Walter
16.1 Introduction . . . 431
16.2 Montgomery Reduction . . . 431
16.3 Montgomery Modular Multiplication . . . 433
16.4 Exponentiation . . . 435
16.5 Space and Time Comparisons . . . 437
16.6 Side Channel Analysis . . . 438
16.7 Frequencies of Conditional Subtractions . . . 440
16.8 Variance in Frequencies and SCA Errors . . . 442
16.9 A Surprising Improvement . . . 443
16.10 Conclusions . . . 445
16.11 Exercises . . . 445
16.12 Projects . . . 446
References . . . 448

17 Randomized Exponentiation Algorithms . . . 451
Colin D. Walter
17.1 Introduction . . . 451

17.2 The Big Mac Attack . . . 452
17.3 Digit Representation and Exponentiation Algorithms . . . 454
17.4 Liardet–Smart . . . 457
17.4.1 Attacking the Algorithm . . . 459
17.5 Oswald–Aigner Exponentiation . . . 460
17.5.1 Attacking the Algorithm . . . 461
17.6 Ha–Moon . . . 462
17.6.1 Attacking the Algorithm . . . 463
17.7 Itoh's Overlapping Windows . . . 464
17.7.1 Attacking the Algorithm . . . 465
17.8 Randomized Table Method . . . 466
17.8.1 Attacking the Algorithm . . . 466
17.9 The MIST Algorithm . . . 467
17.9.1 Attacking the Algorithm . . . 468
17.10 Conclusions . . . 469
17.11 Exercises . . . 469
17.12 Projects . . . 470
References . . . 472

18 Microarchitectural Attacks and Countermeasures . . . 475
Onur Acıiçmez and Çetin Kaya Koç
18.1 Introduction . . . 475
18.2 Overview and Brief History . . . 476
18.3 Cache Analysis . . . 478
18.3.1 Basics of Cache . . . 478
18.3.2 Overview of Cache Attacks . . . 480
18.3.3 A Brief Survey on Cache Analysis . . . 481
18.3.4 Time-Driven and Trace-Driven Attacks . . . 482
18.3.5 Exploiting Internal Collisions in Time-Driven Attacks . . . 483
18.3.6 Access-Driven Attacks . . . 485
18.3.7 Percival's Hyper-Threading Attack on RSA . . . 489
18.4 Branch Prediction Analysis . . . 490
18.4.1 The Concept of Branch Prediction . . . 490
18.4.2 Simple Branch Prediction Analysis . . . 492
18.5 I-cache Analysis . . . 494
18.6 Exploiting Shared Functional Units . . . 496
18.7 Comparing Microarchitectural Analysis Types . . . 497
18.8 Countermeasures for Microarchitectural Analysis . . . 498
18.9 Exercises . . . 499
18.10 Projects . . . 500
References . . . 501

Authors' Biographies . . . 505

Index . . . 513

Acronyms

2DEM	2D-Encryption Mode
ABC	Accumulated Block Chaining
ABL	Arbitrary Block Length
ACM	Association for Computing Machinery
AES	Advanced Encryption Standard
AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AIS	Anwendungshinweise und Interpretationen zum Schema
AIS	Application Notes and Interpretation of the Scheme
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ARK	Add Round Key
ASIC	Application Specific Integrated Circuits
BPA	Branch Prediction Analysis
BPU	Branch Prediction Unit
BTB	Branch Target Buffer
BS	Byte Substitution
CASR	Cellular Automata Shift Register
CBC	Cipher Block Chaining
CCM	Counter with CBC-MAC
CFB	Cipher Feedback
CHES	Cryptographic Hardware and Embedded Systems
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CMAC	Cipher Based MAC
CMC	CBC Mask CBC
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CS	Cipher State

CTR	Counter Mode
CWC	Carter Wegman with Counter
das	digitized analog signal
DE	Disk Encryption
DEA	Data Encryption Algorithm
DEMA	Differential Electromagnetic Analysis
DES	Data Encryption Standard
DFT	Discrete Fourier Transform
DPA	Differential Power Analysis
DRM	Digital Rights Management
DRNG	Deterministic Random Number Generator
DSA	Digital Signature Algorithm
DLP	Discrete Logarithmic Problem
DSS	Digital Signature Standard
EAX	Conventional Authenticated-Encryption Mode
ECB	Electronic Code Book
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EME	ECB Mix ECB
FEAL	Fast Data Encipherment Algorithm
FFSEM	Feistel Finite Set Encryption Mode
FFT	Fast Fourier Transform
FIGARO	Fibonacci Galois Ring Oscillator
FIPS	Federal Information Processing Standard
FIPS PUB	Federal Information Processing Standard Publication
FPGA	Field Programmable Gate Array
FPLD	Field Programmable Logic Device
gcd	Greatest Common Divisor
GCM	Galois Counter Mode
GF	Galois Field
HCH	Hash Encrypt Hash
HCTR	Hash Counter Hash
HECC	Hyperelliptic Curve Cryptography
HEH	Hash ECB Hash
IACBC	Integrity Aware Cipher Block Chaining
IAPM	Integrity Aware Parallelizable Mode
IEEE	Institute of Electrical and Electronics Engineers
IDEA	International Data Encryption Algorithm
IDFT	Inverse Discrete Fourier Transform
IGE	Infinite Garble Extension
iid	independent and identically distributed
IMA	Institute of Mathematics and its Applications
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
KFB	Key Feedback Mode

LFSR	Linear Feedback Shift Register
LNCS	Lecture Notes in Computer Science
LRU	Least Recently Used
LRW	Liskov Rivest Wagner
LT	LaGrande Technology
LUT	Lookup Table
MA	Microarchitectural Analysis
MAC	Message Authentication Code
MC	Mixed Columns
MD	Message Digest
MDC	Manipulation Detection Code
MMX	Multimedia Extension
MSMP	Modified Spectral Modular Product
MULGF	Multiply in Galois Field
MULGF2	Multiply in Galois Field Base 2
NACSIM	National Communications Security Information Memorandum
NACSEM	National Communications Security Emanation Memorandum
NTT	Number Theoretical Transform
NIST	National Institute of Standards and Technology
NPTRNG	Non-Physical Random Number Generator
NSA	National Security Agency
NSTISSI	National Training Standard for Information Systems Security
OCB	Offset Code Book
OEF	Optimal Extension Fields
OFB	Output Feedback
OMAC	One-Key CBC
ONB	Optimal Normal Basis
OS	Operating System
PC	Personal Computer
PCFB	Propagating Cipher Feedback
PEP	Polynomial Hash Encrypt Polynomial Hash
PKC	Public Key Cryptography
PKCS	Public Key Cryptography Standards
PL	Phase Locked Loop
PMAC	Parallelizable Message Authentication Code
PNT	Pseudo Number Transform
PRNG	Physical Random Number Generator
PTRNG	Physical True Random Number Generator
PUF	Physically Unclonable Functions
RAM	Random Access Memory
RAMB	Block RAM
RFID	Radio Frequency Identification Device
RIPEMD	RACE Integrity Primitives Evaluation Message Digest
RISC	Reduced Instruction Set Computer
RMAC	Randomized MAC

RNG	Random Number Generator
RSA	Rivest Shamir Adleman
RSD	Redundant Signed Digit
SBPA	Simple Branch Prediction Analysis
SCA	Side Channel Analysis
SEMA	Simple Electromagnetic Analysis
SFU	Shared Functional Units
SHA	Secure Hash Algorithm
SIAM	Society for Industrial and Applied Mathematics
SIMD	Single Instruction Multiple Data
SIV	Synthetic IV
SME	Spectral Modular Exponentiation
SMM	Spectral Modular Multiplication
SMP	Spectral Modular Product
SMT	Simultaneous Multithreading
SPA	Simple Power Analysis
SPRP	Strong Pseudo Random Permutation
SR	Shift Rows
TDEA	Triple Data Encryption Algorithm
TEMPEST	Transient Electromagnetic Pulse Emanation Standard
TET	Hash ECB Hash
TMAC	Two-Key CBC MAC
TRNG	True Random Number Generator
TXT	Trusted Execution Technology
VCO	Voltage Controlled Oscillator
VHDL	Very High Level Hardware Description Language
VLIW	Very Long Instruction Word
VT	Virtualization Technology
WAIFI	Workshop on the Arithmetic of Finite Fields
XCB	Extended Code Book
XCBC	Extended Cipher Block Chaining
XECB	Extended Electronic Code Book

Chapter 1
About Cryptographic Engineering

Çetin Kaya Koç

1.1 Introduction

Cryptographic engineering is the name we have coined to refer to the theory and practice of engineering of cryptographic systems, i.e., encryption and decryption engines, digital signature and authentication hardware and software systems, key generation, distribution, and management systems, and random number generators. A cryptographic engineer designs, implements, tests, and validates cryptographic systems. She is also interested in cryptanalyzing them for the purpose of checking their robustness and their strength against attacks, and also building countermeasures in them in order to thwart such attacks by reducing their probability of success.

This is a subject barely taught in our undergraduate and graduate schools. Most courses in cryptography deal with theory, generally introducing mathematically expressed algorithms without showing (or knowing) how they are realized in actual software or hardware. As expected, the devil is in the details: The fastest and most practical implementation of the RSA algorithm requires the implementation of Montgomery multiplication. However, the last step in this algorithm (the so-called final subtraction) yields information which allows an attacker capable of observing, recording, and analyzing the timings of the process to learn some of the private bits. One cannot deduce this information by looking at a mathematical description of the RSA algorithm found in a textbook.

Cryptographic engineering material is scattered among many journal and conference papers, and the practitioners are too busy to write books. A group of us got together in Lausanne, Switzerland, in 2002, and began to teach short courses to engineers and researchers from industry and academia. The idea of putting our course notes into a book was born there and then.

City University of Istanbul & University of California Santa Barbara
e-mail: koc@cryptocode.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_1,

© Springer Science+Business Media, LLC 2009

Cryptographic engineering is a fast-moving field. Every year in conferences such as the CHES (Cryptographic Hardware and Embedded Systems) Workshop, new innovative hardware and software realizations of cryptographic algorithms are introduced or new attacks to cryptanalyze these actual hardware and systems are proposed. This explains the unwillingness of researchers in cryptographic engineering to write books; we are more interested in designing new cryptographic systems or breaking the systems designed by our colleagues!

However, people who are new to this exciting field need good introductions. Engineers from industry and students from our colleges and graduate schools can use this book as a first step to cryptographic engineering.

1.2 Chapter Contents

This book has 18 chapters. It can be divided into 4 parts; however, the sections are intimately interconnected and there is a logical construction of the sections starting from the first chapter. There are also chapters which can belong to more than one part, as one might expect.

Chapters 2, 3, and 4 constitute the *first part* of the book. These chapters investigate and uncover the roles of random numbers in cryptography, and propose evaluation methods and practical designs for random number generators. Random numbers are used in other sciences; for example, the so-called Monte Carlo methods use random numbers to simulate physical or mathematical systems. In cryptography, random numbers provide the uncertainty and unpredictability upon which we build the secrecy of our cryptographic keys. For us, their most important property is requirement R2 (see, Chapter 2) which says that the full knowledge of a current bit does not help us to guess its past or future companions better than 50% chance. Chapter 2 examines the general definitions, requirements, and classifications of random numbers while Chapter 3 proposes an evaluation criteria for true random number generators (TRNGs). The ideas behind Chapter 3 produced the world's first evaluation methodology for TRNGs, called AIS.

Chapter 4, on the other hand, proposes a few practical TRNG designs suitable for implementation using ASIC and reconfigurable logic blocks. There is no doubt that, as we improve our understanding of the evaluation of TRNGs, more practical (low power, small circuit area, etc.) TRNG designs will be produced. I believe we are just entering this exciting field of TRNG designs, which requires collaboration by analog and digital circuit designers and cryptographers.

The *second part* of the book (Chapters 5–9) concentrates on implementation (i.e., hardware and software realizations) of public-key cryptographic systems, such as RSA, Diffie-Hellman, and elliptic curve cryptography, and their underlying arithmetic which includes large-integer arithmetic, arithmetic in prime fields and binary extension fields. Chapter 5 gives a general introduction to finite field arithmetic and describes the basic algorithms. Chapter 6 introduces the so-called unified arithmetic (which is also called dual-field arithmetic). The unified arithmetic allows one to

design a single hardware unit with negligible additional cost that performs arithmetic in both $GF(p)$ and $GF(2^k)$.

Chapter 7 introduces a new and compelling research area: the use of discrete Fourier transforms over finite rings in order to design parallel functional units for modular arithmetic. While the use of Fourier transforms to perform fast multiplication is well known, this chapter proposes the first spectral algorithm for modular multiplication.

Chapter 8 provides a high-level, mathematical view of elliptic and hyperelliptic curve arithmetic; it is also a good introduction to vulnerabilities of and attacks on elliptic and hyperelliptic curve cryptography. Finally, Chapter 9 provides a detailed account of instruction set architectures for cryptography, for both secret-key and public-key cryptographic algorithms. Chapter 8 provides a smooth transition from public-key cryptography to secret-key cryptography, a topic which we deal with in the subsequent part of the book.

The *third part* of the book studies implementation aspects of secret-key cryptographic algorithms, which are in Chapters 10, 11, and 12. The emphasis of these chapters is that they concenrate on hardware realizations of secret-key ciphers and their simple (ECB, CBS) and advanced (CCM) modes of operations. Chapter 10 covers both ASIC and FPGA realizations, while Chapter 11 particularly deals with FPGA implementations, exploiting logic structures more efficiently. Chapter 12, on the other hand, is a good summary on modes of operation, with special concentration on modern modes. The most important mode seems to be the CCM mode, which is an authenticated encryption mode used particulary in wireless communication protocols.

The final and *fourth part* of the book is the longest part (Chapters 13–18), and deals with the important topics of side-channel cryptanalysis and countermeasures against such attacks. Chapter 13 gives a brief introduction to the side-channel analysis. It covers the basic principles of side-channel cryptanalysis and introduces simple countermeasures to prevent side-channel leakage.

Chapter 14 delves into more advanced topics and shows how only a fraction of the information obtained from a side-channel can be used to cryptanalyze a practical system. Chapter 15 explains a particular type of side-channel: electromagnetic emanations from physical systems can be collected and analyzed by an attacker in order to capture messages not intended for others to see.

Chapters 16 and 17 show how algorithmic properties can be modeled and utilized to guess the bits of private keys. Chapter 16 focuses on how Montgomery multiplication leaks information, while Chapter 17 introduces methods to make the job of the attacker infeasible by using randomized exponentiations.

Finally, Chapter 18 introduces microarchitectural side-channel attacks, which allow an attacker to obtain information about cryptographic key bits from a crypto-process running on a client or server computer, by sneaking an unprivileged spy process into the same processor. These attacks slightly differ from the classic side-channel attacks, and are shown to be quite effective. It is very likely that future processors will have to be designed with hardware countermeasures against these microarchitectural side-channel attacks.

1.3 Exercises and Projects

Whenever appropriate, a chapter ends with two sections for the purpose of checking the reader's understanding of the chapter's technical material and leading her into research by describing a few doable projects. If the book is used in a graduate-level course, the *exercises* can be given as homework assignments. On the other hand, the *projects* are suitable for small groups (1 or 2 individuals) to implement.

Acknowledgements I would like to express my gratitude to all authors of the chapters in the book. Without their dedication, this book would not have come into existence.

I would also like to thank the staff of Springer US, particularly, Jason Ward, Caitlin Womersley, and Katelyn Stanne for helping me through the steps to completing the book, and also being patient with me and my co-authors, as we struggled to create time for this book from our other duties.

I thank my dear friends Vlado Valence and Caroline Huber of Mead Education, the co-organizers of the EPFL lectures in cryptographic engineering. Above all, I am indebted to Gabor Temes, who encouraged us to organize these lectures in the first place.

I thank my former and current students for contributing to this book by co-authoring chapters, reading and correcting portions of the material, and being life-long collaborators. I would like to thank particularly Gökay Saldamlı, Serdar Erdem, Tuğrul Yanık, Erkay Savaş, and Sultan Selçuk for helping in the creation and correction of the manuscript.

Finally, I thank my family for their constant patience and love.

Copyrights and Permissions A great many of figures and tables, and much of the other material found in this book are from the proceedings of the following conferences and workshops: Cryptographic Hardware and Embedded Systems, Cryptography and Coding, Computational Science and Its Applications, Field-Programmable Logic and Applications, Information Security and Cryptology, Information and Communications Security, CT-RSA, ASIACRYPT, and INDOCRYPT.

The above conference proceedings are published by Springer in the Lecture Notes of Computer Science series. This material is printed in this book with the kind permission of Springer Science+Business Media.

Chapter 2
Random Number Generators for Cryptographic Applications

Werner Schindler

2.1 Introduction

A large number of cryptographic applications require random numbers, e.g., as session keys, signature parameters, ephemeral keys (DSA, ECDSA), challenges or in zero-knowledge protocols. For this reason, random number generators (RNGs) are part of many IT-security products. Inappropriate RNGs may totally weaken IT systems that are principally strong, e.g., if an adversary is able to determine session keys.

It is intuitively clear that random numbers should remain unpredictable, even if an adversary knows a large number of other random numbers (predecessors or successors of the random numbers of interest) that have been generated with the same RNG, e.g., from openly transmitted challenges or session keys from messages that the adversary has received legitimately. Ideally, random numbers should be uniformly distributed on their range and independent. However, this characterizes an *ideal RNG*, which is a mathematical construction.

In Section 2.2 we formulate the general requirements RNGs should have, and in Section 2.3 we divide the entity of 'real-world' RNGs into several classes. The main classes are deterministic RNGs and true RNGs, the latter falling into two subclasses (physical and non-physical true RNGs).

The designer of an RNG is faced with two challenges. First he has to develop an appropriate design and implement it suitably. Especially for true RNGs the second task is usually even more difficult, namely to prove or at least to give strong evidence that the chosen design and the concrete implementation are indeed secure.

The main part of this chapter is devoted to deterministic RNGs (Section 2.4). The basic aspects of true RNGs are addressed in Sections 2.5 and 2.6. Evaluation criteria for physical RNGs are treated intensively in the following chapter. Section 2.7

Bundesamt für Sicherheit in der Informationstechnik
e-mail: Werner.Schindler@bsi.bund.de

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_2,

© Springer Science+Business Media, LLC 2009

addresses important standards and evaluation guidances for RNGs. Sections 2.8 and 2.9 contain exercises and possible implementation projects.

2.2 General Requirements

Many cryptographic applications require random numbers. The protocol usually only demands 'generate a 64-bit challenge', 'generate a random prime', 'generate a random session key' etc., but does not specify any requirements these random values should have. Intuitively, the matter seems to be clear: Random numbers should assume all possible values with equal probability and should be independent from predecessors and successors. However, these (usually unspoken) requirements are very restrictive and characterize an ideal RNG. Note that even if a real-world RNG was ideal it is hardly possible to give evidence in a strict sense (cf. the next chapter).

A closer look at typical applications allows a positive formulation of necessary requirements. Absolutely inevitable is

- (R1) The random numbers should have good statistical properties.

Requirement (R1) is usually checked with a particular statistical test suite, ideally adjusted to the concrete RNG. For specific applications, as for many challenge–response protocols or openly transmitted IVs for block ciphers in CBC mode, (R1) should be fully sufficient. In particular, (R1) shall exclude replay attacks or correlation based attacks.

Unfortunately, (R1) is insufficient for sensitive applications. In Section 2.4.3 we will treat RNGs that have good statistical properties but allow an adversary to predict the whole sequence of random numbers from a small, known subsequence. The assumption that an adversary knows some random numbers is realistic for many applications. Consider, for instance, the generation of session keys if the same RNG is also used for challenges that are transmitted openly. Another example is a classical hybrid protocol where Alice encrypts a confidential message with a randomly selected session key k_{rnd} and sends k_{rnd} to the legitimate receiver, using a suitable key exchange protocol. Of course, the legitimate receiver of particular messages shall not be able to decrypt other messages. In this context, a legitimate receiver of a message is principally a privileged attacker since he knows at least one session key. If Alice represents a public server an adversary may learn millions of random numbers. This suggests the next requirement, namely.

- (R2) The knowledge of subsequences of random numbers shall not allow one to *practically* compute predecessors or successors or to guess these numbers with non-negligibly larger probability than without knowledge of these subsequences.

In Section 2.4 we will introduce two further requirements that are characteristic for DRNGs.

2.3 Classification

Following [1] (which narrows the focus to random bit generators) 'real-world' RNGs fall into two main classes. The first class consists of the *deterministic RNGs* (DRNGs, aka *pseudorandom number generators*). Starting with a *seed*, DRNGs generate pseudorandom numbers algorithmically. The *true RNGs* (TRNGs) form the second class, which falls into two subclasses: *physical TRNGs* (PTRNGs) and *non-physical TRNGs* (NPTRNGs). Physical TRNGs use non-deterministic effects of electronic circuits (e.g., shot noise from Zener diode, inherent semiconductor thermal noise, free-running oscillators) or physical experiments (e.g., time between emissions of radioactive decay, quantum random processes). NPTRNGs exploit non-deterministic events (e.g., system time, hard disk seek time, RAM content, user interaction). So-called *hybrid RNGs* have design elements from both DRNGs and TRNGs. Roughly speaking, the security of a DRNG essentially depends on the computational complexity of possible attacks (→ practical security), while TRNGs rely on the unpredictability of their output (→ theoretical security). We will illuminate this aspect later. Depending on their main 'security anchor' we distinguish between *hybrid DRNGs* and *hybrid* TRNGs (Figure 2.1).

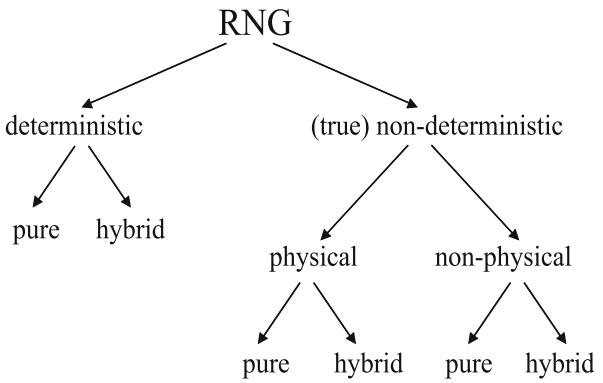

Fig. 2.1 RNG classification.

2.4 Deterministic Random Number Generators (DRNGs)

In this section we consider deterministic random number generators. The main part of this section deals with pure DRNGs but we also consider hybrid DRNGs. We formulate and justify two additional DRNG-specific requirements (R3) and (R4). We illustrate the general principles by many examples, and we also address stochastic simulations and Monte Carlo integration.

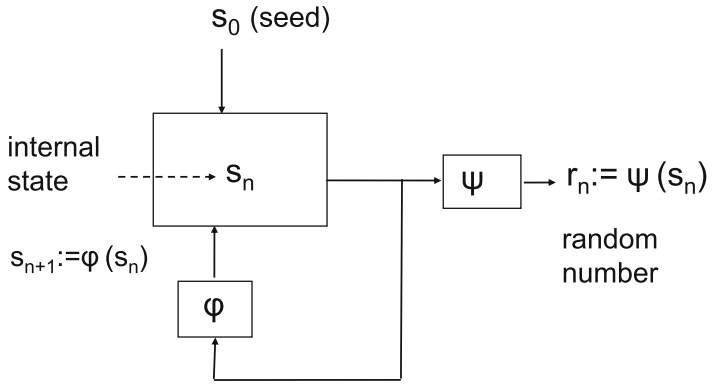

Fig. 2.2 Pure DRNG: Generic design.

2.4.1 Pure DRNGs

This subsection considers the generic design and basic properties of pure DRNGs, and we analyze the security properties of several pure DRNG designs. Figure 2.2 illustrates the generic design of a *pure* DRNG. After $n-1$ random numbers

$$r_1, r_2, \ldots, r_{n-1} \in R$$

have been generated, the internal state of the DRNG attains the value $s_n \in S$. The finite sets S and R are called the *state space* and the *output space* of the DRNG. The *output transition function* $\psi\colon S \to R$ computes the next random number r_n from the current internal state s_n. Then s_n is updated to s_{n+1} with the *state transition function* ϕ, i.e., $s_{n+1} := \phi(s_n)$. The first internal state s_1 is derived from the *seed* s_0, e.g., simply $s_1 = \phi(s_0)$, or a more complicated mechanism may be used. Clearly, the seed s_0 determines all internal states $s_1, s_2, \ldots$ and all random numbers $r_1, r_2, \ldots$. In order to fulfil requirement (R2) the seed must be selected randomly. A pure DRNG can be described by a 5-tuple

$$(S, R, \phi, \psi, p_S) \tag{2.1}$$

where p_S defines the probability distribution of the random seed. Note, however, that the seed generation is performed outside the DRNG boundaries. Usually the seed is generated by a TRNG.

A drawback of DRNGs (compared to TRNGs) is that the output is completely determined by the seed, and the future random numbers depend only on the current internal state. Thus the internal state must be protected even if the device is not active. In particular, implementing a pure DRNG on a PC and using its current internal state in the next session may be dangerous. Typically, DRNGs are implemented on smart cards. Of course, pseudorandom numbers cannot be truly random. On the positive side implementing a DRNG is relatively cheap, and unlike for physical RNGs, no dedicated hardware is needed.

We point out that (R2) demands that the seed entropy must be 'large' and that the state transition function and the output function are sufficiently complex. Pure

DRNGs can at most provide *practical security* (computational security). In an information theoretical sense already a few random numbers fully determine the seed and all the generated random numbers completely. In this regard the situation is similar to that of cryptographic primitives (e.g., to block ciphers). In fact, DRNGs typically apply cryptographic primitives.

Example 2.1. Consider a linear feedback shift register (LFSR) over GF(2) with t cells and recursion formula $a_{n+t+1} \equiv c_1 a_{n+t} + \ldots c_n a_{n+1} \pmod 2$ with $c_1, \ldots, c_n \in \{0,1\}$. In this example, $S = \{0,1\}^t$, $s_n := (a_n, \ldots, a_{n+t-1})$,

$$s_{n+1} = \phi(s_n) = (a_{n+1}, \ldots, a_{n+t}) \ ,$$

$R = \{0,1\}$, $r_n = \psi(s_n) = a_n$.

For primitive feedback polynomials the output sequence $r_1, r_2, \ldots$ is known to have good statistical properties unless t is too small. Hence this DRNG should fulfil (R1). Moreover, LFSRs can be implemented efficiently, and they are very fast. On the other hand, the random numbers $r_1, r_2, \ldots$ depend GF(2)-linearly on the initial state of the LFSR. If an adversary knows about t output bits he can easily recover s_1 and hence the whole sequence $r_1, r_2, \ldots$. Consequently, LFSRs do not fulfil requirement (R2), and they are absolutely inappropriate for sensitive cryptographic applications.

Example 2.2. Assume that Enc: $\{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ defines a block cipher where $\{0,1\}^n$ and $\{0,1\}^m$ denote the plaintext space (respectively, the ciphertext space and the key space). Here $S = \{0,1\}^n \times \{0,1\}^m$ and $R = \{0,1\}^n$. Further, $s_n = (r_n, k)$ where the key k has to be kept secret. Finally, $\psi(r_n, k) = r_n$ and $s_{n+1} = (\mathrm{Enc}(r_n, k), k)$ for $n \geq 0$. For commonly used block ciphers no statistical weaknesses are known, and hence (R1) should be fulfilled.

Now assume that the adversary knows random numbers $r_i, \ldots, r_{i+j}$. Since $r_{t+1} = \mathrm{Enc}(r_t, k)$ the adversary knows $j-1$ (specific) (plaintext ciphertext) pairs. Hence finding r_{i-1} is at least as difficult as a chosen-input attack on Enc. (Actually, the situation is even close to a known plaintext attack.) Analogously, the computation of r_{i+j+1} cannot be easier than a chosen plaintext attack on the decryption function Enc^{-1}. Against strong block ciphers chosen-plaintext attacks on Enc and Enc^{-1} are not practically feasible. (Otherwise these algorithms would not be viewed as secure.) Provided that the seeding process guarantees that k cannot be guessed with non-negligible probability, we may assume that the DRNG fulfils (R2) if Enc=AES or Enc =Triple-DES, for instance.

Example 2.2 demonstrates a typical security proof for DRNGs where security properties are traced back to properties of well-studied primitives. Note that requirement (R2) is not fulfilled for Enc =DES, although in the eighties one would presumably have confirmed this property. This underlines another important property of DRNGs, namely that their assessment may change in the course of time.

Assume that an attacker gets knowledge of the current internal state s_n, e.g., because he has mounted a successful hardware attack on a smart card or has had interim access to a computer where this DRNG is implemented. Of course, then the

random numbers $r_n, r_{n+1}, \ldots$ follow immediately from s_n. For many applications it is desirable that the DRNG additionally meets

- (R3) The knowledge of the internal state shall not allow one to *practically* compute 'old' random numbers or even a previous internal state or to guess these values with non-negligibly larger probability than without knowledge of the internal state.

Requirement (R3) demands one-way state transition functions $\phi : S \to S$. We note that Example 2.2 does not fulfil (R3). Once an adversary knows k he simply decrypts $r_{n-1} = \mathrm{Enc}^{-1}(r_n, k)$, $r_{n-2} = \mathrm{Enc}^{-1}(r_{n-1}, k), \ldots$.

Example 2.3. Let $S = R = \{0,1\}^{160}$ while ϕ and ψ are given by the hash functions SHA-1 and RIPEMD-160. At this time both SHA-1 and RIPEMD-160 are assumed to meet the one-way property. As a consequence, this RNG meets (R3) (as well as (R1) and (R2)).

The next example underlines that not only ϕ and ψ are relevant but also their interaction.

Example 2.4. [weak RNG] Let $S = R = \{0,1\}^{256}$ and $\phi = \psi =$ SHA-256. Obviously, $s_{n+1} = \phi(s_n) = \psi(s_n) = r_n$, and $r_{n+1}, r_{n+2}, \ldots$ follow from r_n. In other words, this RNG does not meet (R2).

Example 2.5. Appendix 3.2 in [2] specifies the generation of pseudorandom ephemeral keys. The 'core' of this algorithm defines a DRNG: $w_0 := f(s_n)$,

$$s' := (1 + s_n + w_0) (\mathrm{mod}\ 2^v) \ ,$$

$w_1 := f(s')$, $s_{n+1} := (1 + s' + w_1) (\mathrm{mod}\ 2^v)$, $r_n := (w_1, w_0)$ with a one-way function f which is defined in [2] (cf. Exercise 2).

We mention that occasionally even DRNGs proposed by adopted standards may contain security flaws. Bleichenbacher detected a weakness in the random number generation specified in the preceding version of [2, 3] which led to a change notice [33]. The problem was the following: Uniformly distributed random numbers r_n on $Z_{2^{160}} := \{0, 1, \ldots, 2^{160} - 1\}$ were transformed to random numbers on Z_p by computing $r_n (\mathrm{mod}\ p)$ where p denotes a 160-bit prime. Obviously, the small values in Z_p occur twice as often as the large ones. Although the weakness itself is obvious, the attack is not. Interestingly, a full paper that describes the attack in detail has never been published. Reference [4] shows that the DRNG that is used by Windows 2000 does not meet requirement (R3) (see also Section 2.6).

Remark 2.1. Many applications provide implicit information on the generated random numbers, e.g., by known (plaintext/ciphertext) pairs that correspond to a random session key. Sometimes less complicated formulae also exist that contain information on the unknown random numbers. For DSA- and ECDSA signatures, for instance, the adversary knows an underdetermined system of linear equations in the

signature key and the ephemeral keys. We point out that this aspect may be more relevant for PTRNGs that aim at security in an information theoretical sense. We will come back to this issue in the next chapter.

A class of DRNGs which is very interesting from a theoretical point of view are *cryptographically secure* RNGs. Their security relies upon intractability assumptions (e.g., that factoring large integers is hard). On the basis of this intractability assumption(s), security properties of the DRNGs, in respect of the random numbers can be proved.

Unfortunately, the security assertions concern the whole family of DRNGs, and in a strict sense, it is usually not clear what this means for a concrete member of this family, i.e., for a fixed DRNG. In this regard, the situation reminds us of DRNGs that rely on the security of (concrete) block ciphers or hash functions (where, not even an asymptotic security proof exists). A drawback of cryptographically secure DRNGs is their low output rate.

Example 2.6. (Blum-Blum-Shub DRNG) Let $n = p_1 p_2$ for two m-bit primes p_1 and p_2 with $p_i \equiv 3 (\mathrm{mod}\, 4)$. Starting with a quadratic residue $x_0 \in Z_n^* := \{0 \leq j < n \mid gcd(j,n) = 1\}$ (seed) we compute $x_{n+1} \equiv x_n^d (\mathrm{mod}\, n)$ and $r_n := x_n (\mathrm{mod}\, 2^{t(m)})$.

The generation of $t(m)$ random bits requires the modular exponentiation of a $2m$-bit integer. It is known that a Blum-Blum-Shub DRNG – or more precisely, a family of Blum-Blum-Shub DRNGs – is asymptotically secure if $t(m) = O(\log \log m)$. Roughly speaking, a non-negligible advantage in guessing the next bit (compared to 'blind guessing') enabled an efficient factoring algorithm, contradicting the factoring intractability.

We mention also that RSA- and Rabin RNGs belong to the class of cryptographically secure RNGs (see [5], Section 5.5, and [6], for instance). We leave this field and refer the interested reader to the relevant literature.

2.4.2 Hybrid DRNGs

Pure DRNGs compute $r_n := \psi(s_n)$ and update their internal state by $s_n \to \phi(s_n)$. Hybrid DRNGs allow additional input from a finite set E_0. The state transition function then reads $\phi_H : S \times E \to S$ with $E = E_0 \cup \{\infty\}$ where ∞ means 'no additional input'. Formally, any pure DRNG can be viewed as a hybrid DRNG with additional input ∞ in each step, i.e., $E = \{\infty\}$ (Figure 2.3).

Example 2.7. Consider Example 2.2 with $E_0 = \{0,1\}^n$ and

$$\phi_H((r_n,k),e_{n+1}) = (\mathrm{Enc}(r_n \oplus e_{n+1}),k)$$

for $e_{n+1} \in E_0$ and $\phi_H((r_n,k),\infty) = (\mathrm{Enc}(r_n),k) = \phi(s_n)$, and $\psi_H = \psi$. (As usual $\oplus$ stands for the bitwise addition modulo 2.)

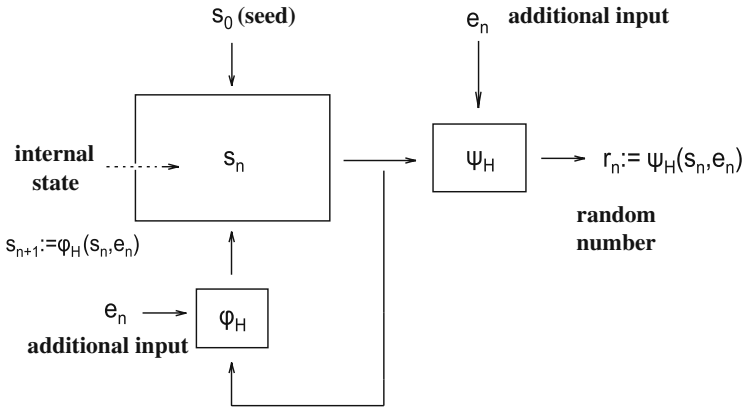

Fig. 2.3 Hybrid DRNG: Generic design.

Of course, a hybrid DRNG cannot be described by a 5-tuple (S,R,ϕ,ψ,p_S) (cf. (2.1)). Instead, we use a 7-tuple

$$(S,R,E,\phi_H,\psi_H,p_S,(q_n)_{n\in N}). \tag{2.2}$$

The set E and the sequence $(q_n)_{n\in N}$ denote the set of additional input data and the probability distributions of the additional data. Note that if $E=\{\infty\}$ and $q_n=\varepsilon_\infty$ (Dirac measure, which has its total mass concentrated on ∞) for all $n\in N$ ('never any additional input') the 7-tuple describes a pure DRNG.

Clearly, even if the input sequence $e_1,e_2,\ldots$ is constant or completely known by a potential attacker this does not reduce the security of the hybrid DRNG below the security of the respective pure DRNG from Example 2.2. Whether the additional input actually increases the security depends on its randomness and unpredictability properties. If the additional input is derived from the current time, for instance, the security gain may be small, depending on the knowledge of the attacker.

For certain applications the following property is desirable, namely when an attacker gets knowledge of the current internal state of the DRNG (e.g., of a software DRNG implementation on a PC) without being noticed by the user of this DRNG, which generates further random numbers.

- (R4) Even the knowledge of the internal state shall not allow one to *practically* compute the next random numbers or to guess these values with non-negligibly larger probability than without the knowledge of the internal state.

Of course, pure DRNGs cannot fulfil (R4). Whether (R4) is met depends on the randomness of the additional input. Regular additional input from a strong TRNG clearly implies (R4). We will learn more about TRNGs in Sections 2.5 and 2.6 and in the next chapter.

In Example 2.7 we *updated* the internal state before applying the seed transition function ϕ. In fact, ϕ_H may be viewed as a two-step procedure. We note that in the

first step the mapping $r_n \to r_n \oplus e_{n+1}$ is injective for any fixed e_{n+1}. This has the pleasant consequence that the security of the hybrid DRNG cannot drop below the level of the respective pure DRNG, regardless of the nature of the additional input and the adversary's knowledge of this input. Much more critical was *reseeding*, realized, for example, by $\phi'_H((r_n,k),e'_{n+1}) = \mathrm{Enc}(r_n,e'_{n+1})$ with $e'_{n+1} \in E' = \{0,1\}^m$. For reseeding, the unpredictability of the additional input is absolutely inevitable.

Remark 2.2. (i) Requirements (R3) and (R4) are specific DRNG requirements. For TRNGs, (R3) and (R4) are usually 'automatically' fulfilled if (R2) is valid.
(ii) In some scenarios the designer may not be able to specify the distributions of the additional input data. Then no security value can be assigned to the additional input. For a seed-update in a strict sense only the security level of the respective DRNG can be assured (provided that $\psi_H(\cdot,e)$ is injective for each $e \in E$). In case of reseeding, no security assertions are yet possible.

Example 2.8. ANSI X9.17 DRNG (hybrid DRNG)
Let $s_n = (r_n,k)$ where k denotes a Triple-DES key while the additional input t_n is a 64-bit representation of the current time. Compute $e_n := \text{Triple-DES}(t_n;k)$, $r_n := \text{Triple-DES}(s_n \oplus e_n;k)$, $s_{n+1} := \text{Triple-DES}(r_n;k)$. This DRNG does not fulfil (R3) if the adversary knows the exact times when random numbers are generated, i.e., if an adversary knows all values of t_n (Exercise 4).

We point out that [1], Annex C, provides several more complex examples of strong DRNGs which use block ciphers, hash functions, or elliptic curves. Reference [1] defines several security levels, demanding different parameter sets.

2.4.3 A Word of Warning

Pseudorandom numbers are used in several branches of applied mathematics, e.g., for stochastic simulations or Monte Carlo integrations. For these applications only statistical properties are significant (cf. [7], for instance) while the unpredictability of pseudorandom numbers is irrelevant. Consequently, pseudorandom numbers that are suitable for stochastic simulations or Monte Carlo integrations usually are completely inappropriate for sensitive cryptographic applications. However, the identical terminology occasionally confuses designers of cryptosystems who have only limited experience with RNGs.

Linear congruential random number generators are widespread since they are extremely fast. Assume, for example, that

$$s_{n+1} \equiv as_n + 1 \pmod{2^m} \quad \text{with } a \equiv 1 \pmod 4. \tag{2.3}$$

Then $x_n := s_n/2^m \in [0,1)$, $n = 1,2,\ldots$ gives a sequence of so-called standard random numbers which are assumed to have similar statistical properties as values taken on by independent random variables $X_1, X_2, \ldots$ that are uniformly distributed on the

unit interval. The sequence $s_0, s_1, \ldots$ is periodic with maximum length 2^m. The parameter $m = 64$ fits perfectly to 64-bit computer architectures. Of course, from s_n a potential attacker can easily determine the whole sequence $x_1, x_2, \ldots$ of pseudo-random numbers. Moreover, the k least significant bit of s_n has period length 2^k. This does not play a role for typical applications of stochastic simulations since there only the most significant bits of the real numbers $x_1, x_2, \ldots$ are relevant. For the generation of secret data short periods of particular bits are not tolerable. Even more, in specific applications short periods might enable correlation attacks. Linear congruential generators do not even meet the basic requirement (R1).

We point out that linear congruential generators can be strengthened in the sense of unpredictablity at the cost of throughput, namely by outputting, let's say, only the m–k most significant bits of s_n, i.e., returning $x_n := (s_n >> k)/2^{m-k}$. However, Knuth found a recovery attack that requires $O(2^{2k} m^2 t^{-2})$ operations if the adversary knows t random numbers $x_1, \ldots, x_t$ ([8]). We mention also that other moduli than powers of 2 and generalizations of linear congruential generators have been studied. We do not delve into this aspect here. We strictly recommend not to use linear congruential generators or related designs for sensitive cryptographic applications.

2.5 Physical True Random Number Generators (PTRNGs)

In this section we explain the generic design of *Physical TRNGs* (PTRNGs) and their important properties. Moreover, we address the concept of entropy and workload. For a thorough treatment of evaluation aspects for PTRNGs we refer the interested reader to the next chapter.

2.5.1 The Generic Design

Just as DRNGs, PTRNGs also are typically implemented in smart cards. Figure 2.4 illustrates the generic design of a physical RNG. The 'core' is the noise source, typically realized by electronic circuits (e.g., using noisy diodes or free-running oscillators) or by physical experiments (radioactive decay, quantum effects of photons, etc.) The noise source generates time-continuous analog signals which are (at least for electronic circuits typically) periodically digitized to binary values at some stage. We call the digitized values *digitized analog signals* or briefly *das random number numbers*. If the das random numbers are binary-valued we also speak of *das bits*. The das random numbers may be algorithmically postprocessed to internal random numbers in order to reduce potential weaknesses. Note that reducing weaknesses (and not simply transforming them into others, (e.g., bias into dependencies) requires data compression which in turn lowers the output rate of the RNG. The algorithmic postprocessing may be memoryless, i.e., it may only depend on the

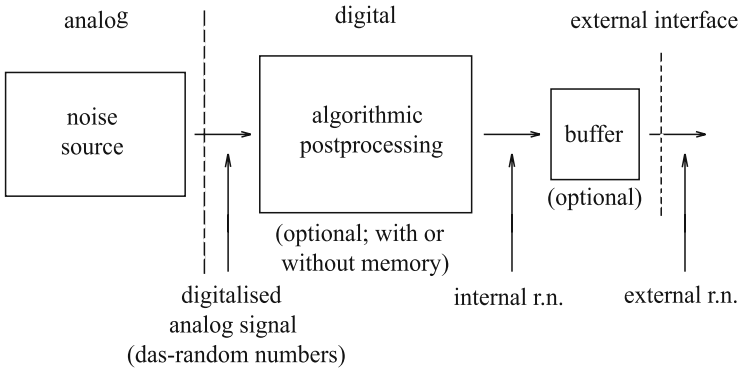

Fig. 2.4 Generic design of a physical RNG.

current das bits, or it may combine the current das random numbers with memory values that depend on the preceding das random numbers (and maybe on some other, possibly secret parameters). Note that strong noise sources do not necessarily require algorithmic postprocessing. Upon external request the internal random numbers are output.

The security of a pure DRNG essentially depends on two factors: on the expected number of guesses to find the seed or any internal state of the DRNG with non-negligible probability and on the complexity of the state transition function and the output function, which determine the workload of possible attacks. Typically, the basic components of DRNGs are cryptographic primitives, and the security of the DRNG can usually be traced back to well-known properties of these primitives (cf. Example 2.2). Clearly, DRNGs can at most be *practically secure*, and their assessment changes in the course of time when attacks on the primitives have become feasible. Consider, for instance, Enc=FEAL-8, which was introduced in the late eighties. Some years later Biham showed that only 2^{24} known-plaintexts are sufficient to recover the 64-bit key. Another prominent example is the change notice in [2] in response to Bleichenbacher's attack.

Contrary to DRNGs, TRNGs (physical and non-physical) rely on the unpredictablity of the generated random numbers. In Section 2.5.2 we will learn that this question is closely related to entropy. Provided, of course, that the entropy estimates are correct, the expected workload to guess such random numbers remains invariant over time. Consequently, it is reasonable to use TRNGs at least for the generation of random numbers that shall protect secrets in the long term. Theoretical security bounds, quantified by the expected number of guesses to find (a sequence of) random numbers, can only be achieved by TRNGs. Unlike for DRNGs, this number does not decrease in the course of time unless, of course, the evaluator made a mistake when estimating the entropy per random bit. We will treat the evaluation of PTRNGs in the next chapter. The following subsection introduces the notion of entropy and guesswork.

2.5.2 Entropy and Guesswork

For the remainder of this subsection, X denotes a random variable that assumes values in the finite set $\Omega = \{\omega_1, \ldots, \omega_m\}$. Without loss of generality, we may assume that $p(\omega_1) := \text{Prob}(X = \omega_1) \geq \cdots \geq p(\omega_m) := \text{Prob}(X = \omega_m)$. The most efficient strategy to guess the outcome of X is clearly to check $\omega_1, \omega_2, \ldots$ until the correct value has been found. The work factor

$$w_\alpha(X) = \min\left\{k : \sum_{i=1}^{m} p(\omega_i) \geq \alpha\right\} \tag{2.4}$$

equals the minimum number of guesses to find the correct value of X with probability $\geq \alpha$. The *guesswork* quantifies the expected number of guesses that are needed to find the outcome of an experiment that is interpreted as a realization of X. The guesswork of X is defined by

$$W(X) := \sum_{j=1}^{m} j p(\omega_j) \tag{2.5}$$

In [9] Shannon introduced the notion of entropy

$$H(X) = -\sum_{j=1}^{m} p(\omega_j) \log_2(p(\omega_j)) \tag{2.6}$$

with $0 \cdot \log_2(0) := 0$. The quantity $H(X)$ is called the *Shannon entropy* or in short, entropy. The most general definition of entropy is the *Rényi entropy* [10], given by

$$H_\alpha(X) = \frac{1}{1-\alpha} \log_2\left(\sum_{j=1}^{m} p(\omega_j)^\alpha\right), \quad 0 \leq \alpha \leq \infty. \tag{2.7}$$

Obviously, $H_\alpha(X)$ is well-defined for $\alpha \neq 1$. For $\alpha = 1$ we set $H_1(X) := \lim_{\alpha \to 1} H_\alpha(X)$. Using L'Hôpital's Rule it is easy to show (Exercise 6) that

$$H_1(X) = H(X). \tag{2.8}$$

Besides $\alpha = 1$, the limit $\alpha \to \infty$ is also of particular importance, which yields the so-called *min-entropy*

$$H_\infty(X) = \min_{j \leq m}\{-\log_2(p(\omega_j))\}. \tag{2.9}$$

Moreover,

$$H_2(X) = \log_2(\text{Prob}(X = Y)) \tag{2.10}$$

with independent, identically distributed random variables X and Y.

For fixed random variable X, the Rényi entropy $H_\alpha(X)$ decreases monotonically for $\alpha \in [0,\infty)$ (Exercise 7), implying that the min entropy is the most conservative entropy measure.

Example 2.9. Let X denote a binary-valued random variable with $\text{Prob}(X=1) = p \in [0,1]$. Then $H(X) = -(p\log_2(p) + (1-p)\log_2(1-p))$, $H_2(X) = -\log_2(p^2 + (1-p)^2)$, and $H_\infty(X) = \min\{-\log_2(p), -\log_2(1-p)\} = -\log_2(\max\{p, 1-p\})$. For $p = 0.5$ (uniform distribution) we have $H(X) = H_2(X) = H_\infty(X) = 1$.

Example 2.10. Assume that X and Y denote binary-valued random variables. For this example we use the abbreviation $q_{xy} := \text{Prob}((X,Y) = (x,y))$. Let $q_{00} = 0.1$, $q_{01} = 0.3$, $q_{10} = 0.3$, and $q_{11} = 0.3$. Elementary computations yield $H(X,Y) = \log_2(10) - 0.9\log_2(3) = 1.895$, $H_2(X,Y) = 1.837$, and $H_\infty(X,Y) = 1.737$.

The work factor $w_{\frac{1}{2}}(X)$ satisfies the following inequality (cf. [11])

$$\left\lfloor 2^{-H_\infty - 1} \right\rfloor \le w_{\frac{1}{2}} \le \left\lceil \left(1 - 0.5 \sum_{j=1}^{m} \left| p(\omega_j) - \frac{1}{m} \right| \right) m \right\rceil . \tag{2.11}$$

If the random variables $X_1, \ldots, X_n$ iid (e.g., describing a memoryless random source) for large n we have $\log_2(w_\alpha(X_1, \ldots, X_n)) \approx nH(X_1)$ for any α ([12], Section 2.3). Note that for the uniform distribution on Ω we have $H_1 = \log_2(k) = H_\infty$, and in the vicinity of the uniform distribution H_1 and H_∞ give similar values.

In the context of PTRNGs, we are usually faced with stationary processes that assume values in $\Omega = \{0,1\}$, for which the entropy per random bit is close to 1. Usually, these processes only have a short-range memory and/or are rapidly mixing. When guessing long sequences of random bits (e.g., session keys) we obtain the same relation between work factor and Shannon entropy as in the memoryless case (cf. Exercise 8).

This justifies the use of the Shannon entropy $H(X)$, which is easier to handle than the Rényi entropy for parameter $\alpha \neq 1$. This, in particular, concerns the conditional entropy which is relevant when evaluating PTRNGs with dependent random numbers.

Let Y be a further random variable that assumes values in a finite set Ω_Y. For any parameter α

$$H_\alpha(X \mid Y = y) = \frac{1}{1-\alpha} \log_2 \left(\sum_{j=1}^{m} \text{Prob}(X = \omega_j \mid Y = y)^\alpha \right), \tag{2.12}$$

defines the conditional entropy if $Y = y$. (If the random variables X and Y are independent, clearly $\text{Prob}(X = \omega_j \mid Y = y) = p(\omega_j)$ for any pair (ω_j, y).) In particular, the conditional Shannon entropy ($\alpha = 1$) equals

$$H(X \mid Y = y) = -\sum_{j=1}^{m} \text{Prob}(X = \omega_j \mid Y = y) \log_2 \text{Prob}(X = \omega_j \mid Y = y) \tag{2.13}$$

The functional equation of the logarithm function, $\log(ab) = \log(a)\log(b)$, implies

$$H(X \mid Y) = \sum_{y \in \Omega_Y} \text{Prob}(Y = y) H(X \mid Y = y), \tag{2.14}$$

representing the conditional Shannon entropy $H(X,Y)$ as a mixture of conditional entropies (2.13) for which $Y = y$ is fixed. There is no pendant to (2.14) for the min-entropy.

2.6 Non-physical True Random Number Generators (NPTRNGs): Basic Properties

Figure 2.5 illustrates the generic design of a non-physical true RNG. The generic design reminds one of PTRNGs, the *entropy source* being the pendant of the noise source. Unlike the noise source of a physical TRNG, the entropy source of an NPTRNG does not require dedicated hardware but exploits system data (e.g., PC time, RAM data, thread numbers, etc.) and/or human interaction (e.g., key strokes, mouse movement). The entropy of these *raw bits* is usually low, demanding a highly compressing postprocessing algorithm. Moreover, the entropy source is not under the designer's control as it depends on the configuration on the computer used and/or the user himself. This implies considerable differences in the security evaluation of PTRNGs and NPTRNGs.

The NPTRNGs are predestined for software implementation on computers. Example 2.11 addresses a typical design. The output of the NPTRNG may be used 'directly', or it may serve to seed (reseed, update the seed of) a DRNG or as additional input. At least at the beginning of a session, the internal state of the DRNG should be updated, which nullifies attacks on the internal state of the DRNG between the particular sessions. To make attacks on the current internal state (e.g., buffer-overflow attacks) inefficient, the internal state of the DRNG should be updated even during the sessions, e.g., periodically after a particular (small) number of internal random numbers have been output.

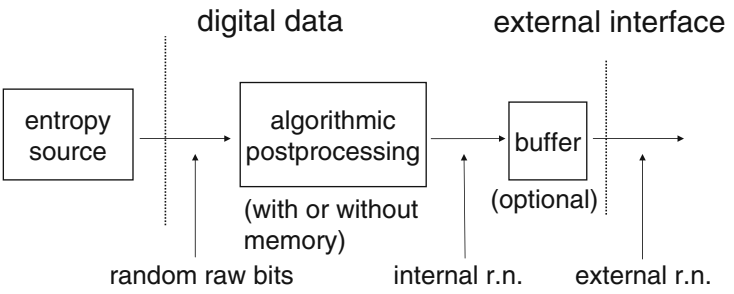

Fig. 2.5 NPTRNG: Generic design.

Example 2.11. Assume that the entropy source uses the system time, the time since system start, several thread numbers and handles, the cursor position and a hash value over a specified RAM area to generate a raw bit string of 1024 bits. This raw bit string is algorithmically postprocessed with the SHA-1 hash function which gives a 160-bit string (internal random number).

Guessing an internal number 'directly' affords 2^{159} trials on average. More interesting is the alternative approach, where the raw bit string is guessed first and then the SHA-1 function is applied to this guess. Unlike in the blind guessing approach, an attacker may exploit his insight into the stochastic properties of the particular components of the raw bit string in order to improve his success probability. We point out that only the one-way property of the SHA-1 hash function is relevant in the context (e.g., when an adversary knows some internal random numbers) whereas collision resistancy is not mandatory. We concentrate on the analysis of a single raw bit string and do not go into detail, but only give general advice.

The central goal of a security evaluation is to ensure that even for small α the work factor w_α is large enough to make guessing attacks infeasible. (The quantitative meaning of 'small' may depend on the intended applications.) To reach this goal the designer, as well as the evaluator, tries to estimate the entropy of the raw bit string. If possible, independent subsets should be identified in order to split the entropy estimation problem into several independent smaller ones. Unlike for PTRNGs, the environments where NPTRNGs run need not essentially be identical but may be very different, which may have impact on the entropy of the raw bits. Unlike for PRTRNGs, it is hardly possible to formulate reliable stochastic models which allow precise entropy estimates.

If the NPTRNG is called automatically when the the PC is booted, at least the time since system start or particular thread numbers should be better predictable (at least for an expert on operating systems) than if the NPTRNG is started on demand. Of course, the concrete implementation, the operating system, and the programs that run on the computer also play a role. From the view of security evaluation, the situation becomes even worse if parts of the raw bit string are derived from the interaction of the user (key strokes, mouse movement). Generally speaking, for NPTRNGs the knowledge of the adversary plays an important role. This may concern technical issues such as the operating system or the used configuration of the attacked system; or the time stamp of an e-mail may provide a rough estimate for the time when the random number was generated. The best the designer or the evaluator of the NPTRNG can do is to determine (and to justify!) a lower entropy bound that shall be valid for all possible implementations and environments, as also for the worst case scenario. Since the raw bits are not stationary, the min-entropy should be the appropriate type of entropy. In practice, normally the Shannon entropy is still used.

Linux operating systems use /dev/random and /dev/urandom to generate random numbers. The function /dev/random may be viewed as a pure NPTRNG while /dev/urandom 'extends' bit strings from the entropy pool, i.e., it may be viewed as a hybrid DRNG that is seeded (and its seed is updated) by an NPTRNG ([13], see also [14]).

We refer the interested reader to [4] which explains a sophisticated attack on the DRNG that is used by Windows 2000. The authors examined the binary code of a particular Windows distribution. This DRNG does not fulfil requirement (R3) since it only requires $O(2^{23})$ operations to get the preceding internal state from the current one. The internal state is periodically updated with an NPTRNG. Since these intervals are too large, and due to the way the random number generator is run by the operating system, a single compromised internal state of the DRNG may compromise up to 128 KBytes of random numbers.

2.7 Standards and Evaluation Guidances

A number of evaluation guidances and standards are effective, and many of them have already been well tried in practice [1, 2, 15–20]. These documents define and explain properties that strong RNGs should have. The evaluation guidances [19] and [20] are technically neutral. They define the criteria and explain how these criteria shall be verified. The criteria are generic in order not to exclude appropriate RNGs. On the other hand, these criteria are clear enough to ensure identical evaluation results, independent of who performs the evaluation. For DRNGs, [19] defines classes K1 to K4 with increasing requirements. Simply speaking, class K2 corresponds to Requirement (R1), while the classes K3 and K4 demand (R1) + (R2) or (R1) + (R3), respectively. Several evaluation guidances and standards [1, 2, 17, 19] give approved designs for DRNGs, or at least discuss examples.

For physical RNGs, it is hardly possible to specify approved designs since secuity-relevant properties depend on the concrete implementation. Of course, particular RNG designs may be analyzed and central steps of the evaluation process defined. Finally, the evaluation yet requires measurements on the concrete implementation. Statistical blackbox tests (as were formulated e.g., in [18, 21]) cannot ensure the security of an RNG.

2.8 Exercises

1. Assume that Alice uses the DRNG from Example 2.2 with Enc =DES to generate 512-bit RSA primes. Discuss this application. Is Alice's choice Enc =DES appropriate?
2. (a) Describe the DRNG from Example 2.5 in our notion. In particular, define the state transition function and the output function.
   (b) Which of the security requirements (R1 to R3) does this DRNG fulfil?
3. Formulate the describing 5-tuple for the Blum-Blum-Shub DRNG (Example 2.6).
4. Example 2.8 describes the ANSI X9.17 hybrid DRNG which has been used in many applications. Assume that an adversary knows all additional input data e_n. Show that this DRNG does not fulfil (R3).

5. Let X denote a random variable that assumes values in $\{0,1\}^{128}$. In particular, $\text{Prob}(X = (0,\ldots,0)) = 0.5$ while $\text{Prob}(X = \omega) = 2^{-128}$ for all strings $\omega \in \{0,1\}^{128}$ that begin with 1. Compute the work factor $w_{0.5}$, the guesswork $W(X)$, the Shannon entropy and the min entropy. Discuss the results.
6. Prove Formula (2.8).
7. Show that the Rényi entropy $H_\alpha(X)$ is monotonically decreasing for $\alpha \in [0,\infty)$.
8. Assume that $X_1, X_2, \ldots$ define a stationary (but not necessarily independent) stochastic process such that the vectors $(X_a,\ldots,X_b)$ and $(X_c,\ldots,X_d)$ are independent if $a < b < c < d$ and $c - b > 1$. Show that $\log_2(w_\alpha(X_1,\ldots,X_n)) \approx H(X_1,\ldots,X_n)$ for any fixed α if n is sufficiently large.
9. Verify Formula (2.14).
10. Consider the NPTRNG from Example 2.11. Assume that the designer adds the absolute time at system start to the raw bit string. Does this increase the security of the NPTRNG? Explain your answer.

2.9 Projects

1. Implement a non-physical true random number generator (NPTRNG) on a PC. Try to state and justify lower entropy bounds. Work out the differences between Windows and Linux operating systems.
2. Implement the DRNG from Exercise 4 in software and determine the data rate that is achievable on your computer.

References

1. ISO/IEC 18031. *Random Bit Generation*. November, 2005.
2. NIST. *Digital Signature Standard (DSS)*. FIPS PUB 186-2, 27.01.2000 with Change Notice 1, 5.10.2001. csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf
3. Lucent Technologies, Bell Labs. *Scientist discovers significant flaw that would have threatened the integrity of on-line transactions*, press article at www.lucent.com/press/0201/010205.bla.html.
4. L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the Windows Random Number Generator. In *Proc. ACM—CCS 2007*, ACM Press, pp. 476–485, New York, 2007.
5. A. J. Menezes, P. C. v. Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, Boca Raton (1997).
6. J. C. Lagarias. Pseudorandom Number Generators in Cryptography and Number Theory. *Proc. Symp. Appl. Math.*, 42: 115–143, 1990.
7. G. Marsaglia. *Diehard* (Test Suite for Random Number Generators). www.stat.fsu.edu/∼geo/diehard.html

8. D. E. Knuth. Deciphering a Linear Congruential Encryption. IEEE Trans. Inform. *Theory*, 31: 49–52, 1985.
9. C. Shannon. Mathematical Theory of Communication. *Bell System Technology*, 27, 1949.
10. A. Rényi. On the Measure of Entropy and Information. In *Proc. Fourth Berkeley Symp. Math. Stat. Prob. 1 1960*, University of California Press, Berkeley, 1961.
11. J. O. Pliam. *The Disparity Between the Work and the Entropy in Cryptology*, 01.02.1999. eprint.iacr.org/complete/
12. J. O. Pliam. Incompatibility of Entropy and Marginal Guesswork in Brute-*Force Attacks*. In B. K. Roy, E. Okamoto editors, *Indocrypt 2000*, Springer, Lecture Notes in Computer Science, Vol. 2177, 67–79, Berlin, 2000.
13. T. Ts'o. random.c—Linux kernel random number generator. http://www.kernel.org.
14. Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the Linux Random Number Generator. IEEE Symp. on Security and Privacy, IEEE, pp. 371–385, 2006.
15. AIS 20. *Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators*. Version 1, 02.12.1999 (mandatory if a German IT security certificate is applied for; English translation). www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf
16. AIS 31. *Functionality Classes and Evaluation Methodology for Physical Random Number Generators*. Version 1, 25.09.2001 (mandatory if a German IT security certificate is applied for; English translation). www.bsi.bund.de/zertifiz/zert/interpr/ais31e.pdf
17. ANSI X9.82. *Random Number Generation* (Draft Version).
18. NIST. *Security Requirements for Cryptographic Modules*. FIPS PUB 140-2, 25.05.2001 and Change Notice 1, 10.10.2001. csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
19. W. Schindler. *Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators*. Version 2.0, 02.12.1999, mathematical-technical reference of [15] (English translation); www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf
20. W. Killmann and W. Schindler. *A Proposal for Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators*. Version 3.1, 25.09.2001, mathematical-technical reference of [16] (English translation); www.bsi.bund.de/zertifiz/zert/interpr/trngk31e.pdf
21. NIST. *Security Requirements for Cryptographic Modules*. FIPS PUB 140-1, 11.04.1994. www.itl.nist.gov/fipspubs/fip140-1.htm
22. M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. SIAM *Journal of computers*, 13 850–864: 1984.
23. J.-S. Coron and D. Naccache. An Accurate Evaluation of Maurer's Universal Test. In S. Tavares and H. Meijer editors. *Selected Areas in Cryptography—SAC '98*. Springer, Lecture Notes in Computer Science, Vol. 1556 pp. 57–71, Berlin, 1999.
24. L. Devroye. Non-Uniform Random Variate Generation. Springer, New York, 1986.

25. U. Maurer. A Universal Statistical Test for Random Bit Generators. *Journal of Cryptology*, 5: 89–105, 1992.
26. A. Rukhin et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800–22 with revisions dated (15.05.2001). csrc.nist.gov/rng/SP800-22b.pdf
27. W. Schindler and W. Killmann. Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications. In B. S. Kaliski Jr., Ç. K. Koç, C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 2002*, Springer, Lecture Notes in Computer Science 2523, pp. 431-449, Berlin, 2003.

Chapter 3
Evaluation Criteria for Physical Random Number Generators

Werner Schindler

3.1 Introduction

In the previous chapter we first addressed general aspects of random number generators (RNGs) that are used for cryptographic applications. We divided the entirety of RNGs into two main classes, the deterministic RNGs (DRNGs) and the true RNGs (TRNGs), the latter falling into two subclasses, the physical RNGs (PTRNGs) and the non-physical true RNGs (NPTRNGs). Moreover, we distinguished between pure and hybrid RNGs, the latter deploying features from both, deterministic and true RNGs.

We formulated four general requirements ((R1) to (R4)). Which of these requirements are necessary depends on the concrete application. Inevitable for sensitive cryptographic applications are

- (R1) The random numbers should have good statistical properties.

  and

- (R2) The knowledge of subsequences of random numbers shall not allow one to *practically* compute predecessors or successors or to guess these numbers with non-negligibly larger probability than without knowledge of the subsequence.

For DRNGs, (R3) and (R4) (backward and forward secrecy) are essentially additional requirements. For TRNGs, (R3) and (R4) usually follow immediately from (R2) if we neglect very specific constructions.

In the previous chapter we treated DRNGs in detail, and we addresed basic facts of PTRNGs and NPTRNGs. The security of DRNGs essentially grounds on the complexity of the state transition function and the output function. In contrast, the security of TRNGs is based on 'true' randomness as their name indicates. DRNGs usually apply cryptographic primitives. Consequently, DRNGs can at the most be

Bundesamt für Sicherheit in der Informationstechnik
e-mail: Werner.Schindler@bsi.bund.de

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_3,
© Springer Science+Business Media, LLC 2009

computationally secure, and this assessment may change in course of time if weaknesses of the used primitives are detected and/or the computational power of a potential adversary increases. Unless the evaluator has made a mistake in the evaluation process, the determined workload to guess 'true' random numbers remains invariant in the course of the years. Consequently, it is reasonable to use TRNGs at least for the generation of random numbers that shall protect secrets in the long term. The connection between entropy and guessing workload was discussed in Section 5.2 of the previous chapter. In the last few years physical RNGs have attracted enormous attention in the scientific community and in the semiconductor industry. A large number of PTRNG designs have been proposed and analyzed ([4, 5, 7, 13, 25, 26, 42] etc.).

This chapter deals exclusively with the security evaluation of PTRNGs. The main task of the evaluator (but not his only one) is to determine the entropy per random bit, or at least a lower entropy bound. The evaluation of physical and non-physical TRNGs is very different. PTRNGs use dedicated hardware which is essentially identical for all devices (apart from tolerances of components or ageing effects). This allows a precise stochastic model of the noise source which is the basis of any sound entropy estimation. In contrast, NPTRNGs are usually implemented on PCs and exploit system data and/or the user's interaction. It is hardly possible to formulate a precise stochastic model that is valid for all implementations of the NPTRNG. Note that these implementations are not under the control of the designer of the NPTRNG (cf. Section 6 in Chapter 2).

At first we repeat central definitions and explain briefly the generic design of a physical RNG. The central goal of any PTRNG evaluation is the estimation of entropy per random bit. To reach this goal we develop general evaluation criteria and introduce the notion of a stochastic model. We investigate the algorithmic postprocessing, and online tests are also discussed in detail. Our expositions are illustrated by many examples and exercises. Further, we compare alternative security paradigms and have a brief look at the standards and evaluation guidances. We close the chapter with some thoughts on side-channel and fault attacks on PTRNGs.

3.2 Generic Design

Figure 3.1 illustrates the generic design of a physical RNG. The 'core' is the noise source, typically realized by electronic circuits (e.g., using noisy diodes or free-running oscillators) or by physical experiments (radioactive decay, quantum effects of photons, etc.) Usually, the noise source generates time-continuous analog signals which are (at least for electronic circuits usually) periodically digitized at some stage. We call the digitized values *digitized analog signals* or briefly *das random number*. Binary-valued das random numbers are also denoted as *das bits*. The das random numbers may be algorithmically postprocessed to internal random numbers in order to reduce potential weaknesses. The algorithmic postprocessing may be memoryless, i.e., it may only depend on the current das random numbers, or

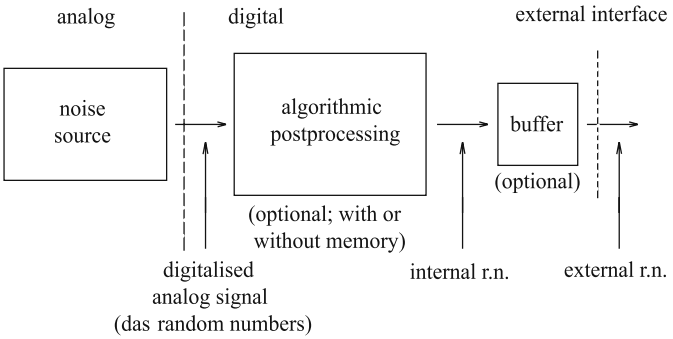

Fig. 3.1 Generic design of a physical RNG.

combine the current das random numbers with memory values that depend on the preceding das random numbers (and maybe some other, possibly secret parameters). Strong noise sources do not necessarily require algorithmic postprocessing. Upon external request internal random numbers are output (→ external random numbers).

3.3 Evaluation Criteria for the Principle Design

The primary goal of a PTRNG evaluation is the estimation of the entropy per random bit, or more precisely, the gain of entropy per random bit. A second task, which yet is not less important, is the evaluation of the online test (cf. Section 3.6). Depending on the intended conditions of use (and the concrete PTRNG design) it may be reasonable to implement explicit countermeasures against fault attacks. Such countermeasures (e.g., shielding) are clearly also part of an overall security evaluation (cf. Remark 3.10).

In Figure 3.1 three types of random numbers occur: das random numbers, internal random numbers and external random numbers. The first question is which of these random numbers should be considered. Of course, finally the quality of external random numbers is relevant since they are used in applications. However, the external random numbers are neither under the control of the RNG designer nor of the evaluator. Fortunately, the external random numbers are usually obtained by concatenating internal random numbers and hence share their statistical properties.

Unfortunately, entropy cannot be measured as voltage or temperature. Entropy is a property of random variables and not of sequences of observed values which are assumed by these random variables. As a first consequence, entropy cannot simply be guaranteed by applying any collection of statistical blackbox tests. Note that usually even pseudorandom sequences pass those blackbox test suites (cf. e.g., [27, 31, 32, 38] developed for a different purpose, namely for testing pseudorandom numbers for stochastic simulations), even if they have been generated by (in our sense) weak DRNGs.

Remark 3.1. The adjective 'universal' in the title of [28] caused a lot of misunderstanding and confusion in the past. Maurer's test segments large-bit streams into non-overlapping blocks of equal length. If the block size tends to infinity, Maurer's test value yields an estimator for the increase of entropy per random bit *provided that the random bits were generated by a stationary binary-valued ergodic random source with finite memory* (cf. also [11]).

If this assumption is not fulfilled Maurer's test value need not have any relation to the entropy per block. If the random numbers were generated by an LFSR, for instance, the increase of entropy per pseudorandom bit is obviously zero while Maurer's test value will be close to the maximum value. The same is true for Coron's test [10].

Instead, to quantify the entropy per random bit we first have to study the distribution of the random numbers, or more precisely, the distribution of the underlying random variables.

Definition 3.1. Random variables are denoted with capital letters. *Realizations* of these random variables, i.e., values assumed by these random variables, are denoted by the respective small letters. A binary-valued random variable is said to be $B(1,p)$-distributed if the values 1 and 0 are assumed with probability p and $1-p$, respectively. As usual, 'iid' abbreviates 'independent and identically distributed'.

The term $H(X)$ denotes the Shannon entropy of the random variable X while $H_\alpha(X)$ denotes its Rényi entropy to parameter α.

We interpret the das random numbers $r_1, r_2, \ldots$ as realizations of random variables $R_1, R_2, \ldots$ and the internal random numbers $y_1, y_2, \ldots$ as realizations of (suitably defined) random variables $Y_1, Y_2, \ldots$.

Remark 3.2. (i) For simplicity, we speak loosely of the entropy per random number or per random bit. What we really mean is the entropy of the underlying random variables.
(ii) In the following we use the term 'adversary'. This includes potential attackers but also the evaluator and the designer of an RNG.

In Section 5.2 of Chapter 2 we explained the relation between entropy and guesswork. We mentioned the min entropy is the most conservative entropy measure, which can be used to obtain universal lower guesswork bounds. However, if the random variables are iid or at least stationary with finite memory, the Shannon entropy quantifies the average guesswork at least asymptotically, i.e., for 'long' sequences of random numbers (e.g., session keys; cf. Chapter 2, Exercise 7). At least the internal random numbers are usually stationary with entropy per bit close to 1. Since the Shannon entropy is easier to handle than the min-entropy ($\rightarrow$ conditional entropy) we will focus on the Shannon entropy in the following. If it is unambiguous we follow the usual convention and briefly speak of 'entropy'.

The term

$$H(Y_{n+1} \mid Y_1 = y_1, \ldots, Y_n = y_n) \tag{3.1}$$

quantifies the entropy of the random variable Y_{n+1} if the adversary knows the internal random numbers $y_1,\dots,y_n$. This corresponds to the real-life situation where an adversary knows a subsequence $y_1,y_2,\dots,y_n$ of internal random numbers, e.g., from openly transmitted challenges or from the session keys of messages which he has received legitimately. The conditional entropy of consecutive internal random numbers (e.g., used to generate random session keys) equals

$$H(Y_{n+1},\dots,Y_{n+t} \mid Y_1 = y_1,\dots,Y_n = y_n) = \sum_{i=1}^{t} H(Y_{n+i} \mid Y_{n+1},\dots,Y_{n+i-1},Y_1 = y_1,\dots,Y_n = y_n). \tag{3.2}$$

Remark 3.3. (i) The conditional entropy (3.1) depends on the knowledge of the adversary (expressed by its conditional part) and quantifies his/her uncertainty on the next random variable.
(ii) For sensitive cryptographic applications (as for the generation of session keys, signature parameters or ephemeral keys) the conditional entropy per bit should be close to 1.
(iii) Note that entropy and conditional entropy may differ considerably. Assume, for instance, that X_1 is $B(1,0.5)$-distributed and $X_1 = X_2 = \cdots$ (total dependency). Then $H(X_n) = 1$ but $H(X_{n+1} \mid X_n) = 0$ for all $n \in \mathbb{N}$.

3.4 The Stochastic Model

The random variables $R_1,R_2,\dots$ and $Y_1,Y_2,\dots$ quantify the stochastic behavior of the das random numbers and the internal random numbers, respectively. These distributions clearly depend on the noise source and the digitization mechanism for the Y_j also on the algorithmic postprocessing algorithm. Usually, the evaluator is not able to determine these distributions exactly. Moreover, in a strict sense, the exact distribution depends on the components of the particular noise source which may differ to some extent even for PTRNGs from the same production series. Instead, as explained below, the evaluator shall specify a *family of distributions* and give evidence that the true distribution of the random numbers (as well as of specific auxiliary random variables) is always (i.e., for all copies of this PTRNG and under all conditions of use) contained in this family.

Example 3.1. (Repeated tossing with the same coin)
We interpret 'head' as 1 and 'tail' as 0, and we denote the generated (tossed) random numbers by $r_1,r_2,\dots$. For the moment we assume that no algorithmic postprocessing is applied, i.e., that $y_i = r_i$ for all i.

Since coins have no memory it is reasonable to assume that the random variables $R_1,R_2,\dots$ are iid binomially $B(1,p)$-distributed with unknown parameter $p \in [0,1]$. This means that the true distribution is contained in a one-parameter family of probability distributions (products of $B(1,p)$-distributions).

For a fixed coin an estimate $\widetilde{p}$ of the true parameter p can be achieved by tossing this coin N times and setting $\widetilde{p} := \#heads/N$.

For 'real-life' PTRNGs the situation is usually more complicated, and the specified family of distributions may depend on several parameters. It is usually reasonable to confirm the specified family of distributions by experiments. In Example 3.1 this may include tests for independency (although in this elementary example additional experiments seem to be superfluous). Generally speaking, the better the analog part of the PTRNG is understood the less experimental verification is necessary.

Our final goal is at least to determine a lower bound for the average entropy per internal random number. Ideally, the *stochastic model* of the noise source comprises the distribution of the internal random numbers, the das random numbers or at least the distribution of auxiliary random variables which allow one to calculate the entropy of the das random numbers or the internal random numbers, or at least to verify lower entropy bounds. We point out that a stochastic model is *not* a physical model of the noise source and its digitization mechanism.

In Example 3.1 a physical model would comprise the mass distribution within the coin and maybe complicated formulae that describe the trajectories of tossed coins (which would be a difficult task). Of course, the stochastic model also depends on the concrete noise source but considers only the impact on the distribution of the random numbers. In particular, we cannot expect that the stochastic model provides an explicit formula for the distribution of the das random numbers (and finally of the internal random numbers) in dependency of the characteristics of the components of the analog part of the noise source, which a physical model might achieve. Instead, we only get a *family of distributions* that depends on one or several parameters.

To reach our formulated goal, the estimation of entropy, we proceed as follows:

1. Formulate a stochastic model for the concrete PTRNG. Justify this model. Try to confirm your assumptions experimentally if there is no clear theoretical proof.
2. Use this PTRNG to generate random numbers. Estimate the unknown parameters that belong to this PTRNG on the basis of these random numbers.
3. Use these parameter estimates to estimate the increase of entropy per random bit.

Example 3.2. (Continuation of Example 3.1)
Recall that for repeated coin tossing the random variables $R_1, R_2, \ldots$ were assumed to be independent. If the sample size N in Example 3.1 was sufficiently large we conclude for any history $r_1, \ldots, r_n$

$$H(R_{n+1} \mid R_1 = r_1, \ldots, R_n = r_n) = H(R_{n+1}) \tag{3.3}$$
$$\approx -(\widetilde{p}\log_2 \widetilde{p} + (1-\widetilde{p})\log_2(1-\widetilde{p})) := \bar{H}(\widetilde{p}). \tag{3.4}$$

For small N (only a few coin tosses) it may be reasonable to determine a confidence interval $I(p)$ for the true parameter p (i.e., an interval that contains p with probability $\geq 1-\alpha$) and estimate the entropy for the least favorable parameter $p' \in I(p)$, providing a lower entropy bound for R_{n+1} (with probability $\geq 1-\alpha$).

Example 3.3. Assume that the random variables $R_1, R_2, \ldots$ form an ergodic homogeneous Markov chain on a finite state space $\Omega = \{\omega_1, \ldots, \omega_m\}$ with transition matrix $P = (p_{ij})_{1 \le i,j \le m}$ ([15], Section XV). As usual, $p_{ij} := \text{Prob}(R_{n+1} = \omega_j \mid R_n = \omega_i)$.

As $R_1, R_2, \ldots$ is an ergodic Markov chain, regardless of the distribution of R_1 the distributions of the random variables $R_1, R_2, \ldots$ converge exponentially fast to a unique limit distribution ν (the unique left eigenvector of P to the eigenvalue 1). If $p_{i.}$ denotes the ith row of P

$$H(R_{n+1} \mid R_1, \ldots, R_n) = H(R_{n+1} \mid R_n) = \sum_{i=1}^{m} \nu(i) H(p_{i.}), \tag{3.5}$$

at least in the limit $n \to \infty$. If the PTRNG is in equilibrium state when r_1 is generated, (which should be the case shortly after the start of the PTRNG), the random variables $R_1, R_2, \ldots$ may be assumed to be ν-distributed, and hence (3.5) is valid for all $n \in \mathbb{N}$ (see also Exercise 1).

Definition 3.2. A sequence of random variables $X_1, X_2, \ldots$ is called *stationary* if for any integer r the distribution of the random vector $(X_{t+1}, \ldots, X_{t+r})$ does not depend on the shift parameter t (cf. Remark 3.4). A sequence $X'_1, X'_2, \ldots$ is said to be q-dependent if the random vectors $(X'_a, \ldots, X'_b)$ and $(X'_c, \ldots, X'_d)$ are independent whenever $c - b > q$. The term $N(\mu, \sigma^2)$ denotes the normal distribution with mean μ and variance σ^2. The cumulative distribution function of the standard normal distribution $N(0,1)$ is denoted with $\Phi(\cdot)$, i.e., $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-0.5t^2}\, dt$

Remark 3.4. Some authors denote stochastic processes that meet the stationarity conditions from Definition 3.2 as *strictly stationary* to distinguish them from weakly stationary (aka weak-sense stationary or wide-sense stationary) stochastic processes. A stochastic process $X'_1, X'_2, \ldots$ is called weakly stationary if the expectations $E(X_j)$ and $E(X_j X_{j+\tau})$ do not depend on the index j.

For most PTRNG designs it is reasonable to assume that the sequence $R_1, R_2, \ldots$ is stationary, at least within time intervals that are large relative to the output rate of the PTRNG. (For example, if the PTRNG generates 1 million random bits per second, a minute or even an hour can doubtlessly be viewed as a 'long' time interval.) Long-term shifts of parameters in the life cycle of a PTRNG (e.g., due to ageing effects) are tolerable *if the distribution remains in the acceptable part of the specified family of distributions*. Note that it is usually very difficult to analyze non-stationary stochastic processes and, in particular, to get numerical results.

The distribution of $R_1, R_2, \ldots$ clearly determines the distribution of $Y_1, Y_2, \ldots$. However, complex algorithmic postprocessing may prevent concrete formulae for the Y_j. If the entropy per das random number is already sufficiently large, it may be sufficient to verify that the algorithmic postprocessing does not reduce the entropy per bit. Recall that in the end we are essentially interested in the entropy and not necessarily in the exact distribution of the Y_j (although the latter is cleary favorable).

Example 3.4. Assume that the random variables $R_1, R_2, \ldots$ are iid $B(1,p)$-distributed and that the algorithmic postprocessing algorithm encrypts consecutive, non-overlapping blocks of 128 das bits with a secret AES key k.

Since k is unknown, it is infeasible to specify the distribution of the random variables $Y_1, Y_2, \ldots$. However, the Y_j are iid, and since the postprocessing is injective,

$$H(Y_{n+1} \mid Y_n) = H(Y_{n+1}) = 128H(R_1). \tag{3.6}$$

(Of course, unless $p = 0.5$ the components of the Y_n are not independent.)

Although we are finally interested in the distribution (or at least in the entropy) of the random variables $Y_1, Y_2, \ldots$, we recommend generally to consider the random variables $R_1, R_2, \ldots$ first. The impact of the algorithmic postprocessing should be analyzed in a second step (see also Section 3.5). That is, in a first step we consider the conditional entropies

$$H(R_{n+1} \mid R_1 = r_1, \ldots, R_n = r_n) \text{ for any } r_1, \ldots, r_n, \text{ or at least} \tag{3.7}$$
$$H(R_{n+1} \mid R_1, \ldots, R_n). \tag{3.8}$$

The term (3.8) is the weighted average of (3.7) over all histories $r_1, \ldots, r_n$. Note that (3.8) is often easier to compute than (3.7). However, if (3.8) is close to the maximum value $\log_2(\text{range}(R_j))$, (3.7) may differ significantly from the average value of (3.8) only for a small fraction of histories. If non-negligible differences occur for different histories (or at least, if one expects such differences) it is reasonable to apply algorithmic postprocessing (see Section 3.5). This algorithmic postprocessing should compress the input data or at least 'mix' them.

Example 3.5. Figure 3.2 shows an RNG design that was proposed at CHES 2002 ([44]). In [42] a stochastic model was developed and analyzed, and a lower entropy bound for the internal random numbers was derived. Numerical examples were given. In the following section we address the central aspects.

The noise source consists of two independent ring oscillators that clock a 43-bit LFSR with a primitive feedback polynomial and a 37-bit linear cellular automaton shift register (CASR), respectively. The frequencies of the ring oscillators are not controlled and drift with variations in temperatures and voltage. The states of the

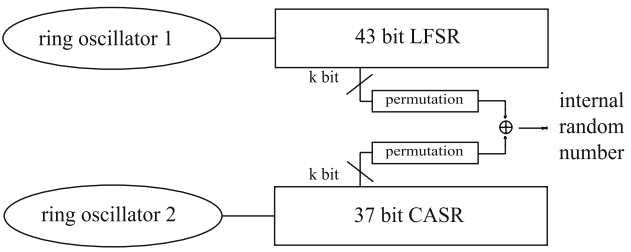

Fig. 3.2 Physical RNG presented at CHES 2002 ([44]).

LFSR and CASR are not reset after power-up. Upon external request $k = 32$ particular cells of both the LFSR and CASR are masked out, permutated and XOR-ed, and this 32-bit XOR-sum is output (internal random number). Even when no random numbers are requested the PTRNG is active. In [44] the minimum time between two consecutive outputs of internal random numbers is specified (LFSR and CASR should at least be clocked 86, and 74 times respectively).

Internal random numbers are output at times $s_0 < s_1 < \ldots$ where r_n is output at time s_n. The das random numbers are given by vectors $(r_{1(1)}, r_{1(2)}), (r_{2(1)}, r_{2(2)}), \ldots$ where $r_{j(1)}$ and $r_{j(2)}$ equals the number of cycles of ring oscillator 1 and 2, i.e., the number of clocks of the LFSR, as well as of the CASR, within the time interval $(s_{n-1}, s_n]$.

Since the oscillators are assumed to be independent we may treat them independently. (In a 'real' PTRNG evaluation the independence assumption requires a theoretical justification and eventually also experimental confimation.) Further, let $t_{i(1)}$ and $t_{i(2)}$ denote the lengths of the ith cycle of ring oscillator 1 and 2, respectively. In [42] we assumed that the corresponding random variables

$$T_{1(j)}, T_{2(j)}, \ldots \quad \text{are stationary} \tag{3.9}$$

(but not necessarily independent) for $j = 1, 2$. Further, let $z_{n(i)}$ denote the smallest index m for which $T_{1(i)} + T_{2(i)} + \ldots + T_{m(i)} > s_n$. With this notation

$$R_{n(i)} = Z_{n(i)} - Z_{n-1(i)} \qquad \text{for } i = 1, 2. \tag{3.10}$$

Example 3.6. Assume that the output voltage of a Zener diode is filtered, amplified and then input to a comparator. The comparator switches whenever the input voltage crosses a specified threshold from below (0-1-crossing). Each switching inverts the D input of a flip-flop. The flip-flop is latched by an external clock with constant period length $s > 0$.

We interpret the number of switchings in time interval $((n-1)s, ns]$ as das random number r_n. In this notation $y_n = y_0 + r_1 + \cdots + r_n \pmod 2$ where $y_0 \in \{0, 1\}$ is the internal random number at time $s = 0$. Let $t_1, t_2, \ldots$ denote the times between consecutive 0-1-switchings. Using the notation from Example 3.5 we conclude

$$R_n = Z_n - Z_{n-1}. \tag{3.11}$$

Also, for this example it is reasonable to assume that the random variables $T_1, T_2, \ldots$ are stationary.

Interestingly, although very different in their technical realization, Examples 3.5 and 3.6 lead to the same type of stochastic model for the das random numbers, and also an RNG construction with two noisy diodes which is intensively studied in [24] meets the following equations

$$T_1, T_2, \ldots \quad \text{are stationary} \tag{3.12}$$

$$R_n := Z_n - Z_{n-1} \quad \text{for} \tag{3.13}$$

$$Z_n := min_{m \in N}\{T_0 + T_1 + T_2 + \ldots + T_m > s_n\} \tag{3.14}$$

where T_0 denotes some 'offset' (the first switching, end of the first ring oscillator cycle etc. after time s_0). (In [42] we assumed $T_0 \equiv 0$ for simplicity, which had little impact in that scenario.) Note, however, that even if the stochastic models of two PTRNGs meet (3.12) to (3.14) the distributions of the random variables $T_1, T_2, \ldots$ and thus of $R_1, R_2, \ldots$ and $Y_1, Y_2, \ldots$ may yet be very different in both cases. Anyway, it is worth analyzing (3.12) to (3.14), both for very mild assumptions on the T_j and for specific classes of distributions (e.g., for iid T_j). We mention some fundamental results. For proofs, details and further assertions the interested reader is referred to [42] and [24].

For $u \geq 0$ we define

$$V_{(u)} := \inf\left\{\tau \in \mathbb{N} \mid \sum_{j=1}^{\tau+1} T_j > u\right\} = \sup\left\{\tau \in \mathbb{N} \mid \sum_{j=1}^{\tau} T_j \leq u\right\}. \tag{3.15}$$

Straightforward considerations lead to

$$\begin{aligned} &\mathrm{Prob}(V_{(u)} = k) = \mathrm{Prob}\,(T_1 + \cdots + T_k \leq u) - \mathrm{Prob}\,(T_1 + \cdots + T_{k+1} \leq u) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } k \geq 1 \\ &\mathrm{Prob}(V_{(u)} = 0) = 1 - \mathrm{Prob}\,(T_1 \leq u), \qquad \text{and} \qquad \mathrm{Prob}(V_{(u)} = \infty) = 0. \end{aligned} \tag{3.16}$$

Let $\mu := E(T_1)$ and $\sigma_T^2 := \mathrm{Var}(T_1) < \infty$. The term

$$\sigma^2 = \sigma_T^2 + 2\sum_{i=2}^{\infty} E\left((T_1 - \mu)(T_i - \mu)\right) \tag{3.17}$$

denotes the *generalized variance* of the random variables $T_1, T_2, \ldots$.

Assume further that $E|T_1^3| < \infty$. If the absolute values of the summands in (3.17) decrease 'rapidly' to zero then a version of the Central Limit Theorem for dependent random variables holds, i.e.,

$$\mathrm{Prob}\left(\frac{\sum_{j=1}^{k} T_j - k\mu}{\sqrt{k}\sigma} \leq x\right) \to_{k\to\infty} \Phi\left(\frac{x - k\mu}{\sqrt{k}\sigma}\right) \tag{3.18}$$

(for details, see [17, 21], for instance). This is in particular the case if the T_j are q-dependent since then the summands in (3.17) are identical zero for $i > q$. For $u = v\mu$ with $v >> 1$ we get the approximation

$$\mathrm{Prob}\left(V_{(v\mu)} = k\right) \approx \Phi\left(\frac{v-k}{\sqrt{k}} \cdot \frac{\mu}{\sigma}\right) - \Phi\left(\frac{v-(k+1)}{\sqrt{k+1}} \cdot \frac{\mu}{\sigma}\right) \quad \text{for } k \geq 1 \tag{3.19}$$

$$\mathrm{Prob}\left(V_{(v\mu)} = 0\right) \approx 1 - \Phi\left((v-1)\frac{\mu}{\sigma}\right). \tag{3.20}$$

By (3.19) and (3.20) the distribution of the random variable $V_{(v\mu)}$ depends only on the ratios μ/σ and $u/\mu = v$ but not on the absolute values of the parameters μ, σ^2, u. Note that the mass of $V_{(v\mu)}$ is essentially concentrated on those k's with $k \approx v$. Since

range$(V_{(u)}) = \mathbb{N}_0$, we extend the definition of Shannon entropy to countable Ω. For an $\mathbb{N}_0$-valued random variable X we define

$$H(X) = -\sum_{i=0}^{\infty} \text{Prob}(X=i)\log_2\left(\text{Prob}(X=i)\right) \quad . \tag{3.21}$$

We point out that $H(X) = \infty$ is possible ([42], Remark 2). In our context, the entropy is always finite (cf. Lemma 2(ii) in [42] which covers the q-dependent case). In particular, (3.19) and (3.20) imply

$$H(V_{(u)}) = -\sum_{i=0}^{\infty} \text{Prob}\left(V_{(u)=i}\right)\log_2\left(\text{Prob}\left(V_{(u)=i}\right)\right) \quad . \tag{3.22}$$

Intuitively, one expects that $H(V_{(u)})$ increases as u increases, although not necessarily monotonously. Note that if $\sigma^2 \approx 0$ then $H(V_{(k\mu)}) \approx 1$ but $H(V_{((k+0.5)\mu)}) \approx 0$ for small integers k, which explains the second statement. We may yet expect $H(V_{(u+\mu)}) > H(V_{(u)})$ since the positions of u and $u+\mu$ relative to the lattice points $\mu, 2\mu, \ldots$ are identical but more 0-1-crossings imply a larger variance of $V_{(\cdot)}$. Of course, this heuristic argumentation is not a strict mathematical proof (we work on it) but explicit computations for a large number of values u support our conjecture.

Since

$$R_n := \sup\left\{\tau \in \mathbb{N} \mid \sum_{j=1}^{\tau} T_{z_{n-1}+j} \le s_n - \sum_{j=1}^{z_{n-1}} T_j\right\} + 1 \tag{3.23}$$

the distribution of R_n is closely related to the distribution of the random variables $V_{(u)}$ considered above. Our goal is to determine (at least a lower bound for) the conditional entropy $H(R_{n+1} \mid R_1, \ldots, R_n)$. With regard to the preceding $H(V_{(s_{n+1}-s_n)})$ might be used as an estimate for this conditional probability, at least if the term $(s_{n+1}-s_n)/\mu$ (= expected number of T_j's within the interval $(s_{n+1}-s_n]$) is large and if $u \mapsto H(V_{(u)})$ only has little variation within an environment of $s_{n+1}-s_n$ (cf. Example 3.5 and [42]). Under these conditions one may hope that an eventual influence of $R_1, \ldots, R_n$ on (the first elements of) $T_{z_n+1}, T_{z_n+2}, \ldots$ has no significant impact on the entropy. To be on the safe side, one may try to (over-)compensate these effects by applying the more conservative entropy estimate

$$\min\{H(V_{(u)}) \mid u \in [s_{n+1}-s_n-a\mu, s_{n+1}-s_n]\} \cdot \text{Prob}(W_n \le a) \tag{3.24}$$

(with moderate $a > 0$) instead of $H(V_{(s_{n+1}-s_n)})$ where

$$W_n := T_0 + T_1 + \cdots + T_{z(n)} - s_n > 0 \quad . \tag{3.25}$$

Ideally, the parameter a should be selected with regard to the distribution of the T_j ($\rightarrow$ dependencies!) and the aimed entropy bound. At least for small ratios μ/s more sophisticated methods are recommendable that take the concrete distribution of the random variables T_j into account, since even moderate parameters a waste information. This demands larger interval lengths $(s_{n+1}-s_n)$ which in turn decreases the output rate of the PTRNG.

For a more sophisticated analysis we concentrate on equidistant instants $s_0, s_1, s_2, \ldots$, i.e., $s_n = ns$ for all $n \in \mathbb{N}$. Recall that the random variables $T_1, T_2, \ldots$ are assumed to be stationary (3.12). (This corresponds to the real-world situation that the noise source is in equilibrium state, which should be the case shortly after starting the PTRNG.) Under mild assumptions (essentially, the partial sums $(T_j + \cdots T_{j+\tau}) (\mathrm{mod}\, s)$ shall be uniformly distributed on $[0,s)$ for 'large' τ) it can be shown that the stochastic processes $R_1, R_2, \ldots$, $Y_1, Y_2, \ldots$, and $W_1, W_2, \ldots$ are stationary, too [24]. In particular, if G_W denotes the cumulative distribution function of W_n then

$$E((R_1 + \cdots + R_j)^k) = \int_0^{js} E((V_{(js-u)} + 1)^k \mid W_0 = u)) \, G_W(du) \tag{3.26}$$

$$\approx \int_0^{js} E((V_{(js-u)} + 1)^k) \, G_W(du) \quad \text{for each } k \in N. \tag{3.27}$$

For iid random variables T_j the $\approx$ sign is in fact $=$ while for dependent T_j the condition '$W_0 = u$' may influence the distribution of the first elements of the $T_1, T_2, \ldots$ via the conditional random variables $(\ldots, T_{-1}, T_0 \mid W_0 = u)$. If the T_j are Markovian, $(T_0 \mid W_0 = u)$ determines the initial distribution of the Markov process $T_0, T_1, \ldots$. Anyway, for 'large' indices j the influence of the condition '$W_0 = u$' on the integral should be negligible. Since $E(R_1 + \cdots + R_j) = jE(R_1)$ applying (3.27) for the parameters $(k = 1, j = 1)$ and, let's say, $(k = 1, j = 10)$ may serve as an indicator for the impact of W_0.

The stationarity of $R_1, R_2, \ldots$ implies

$$E((R_1 + \ldots + R_j)^2) = jE(R_1^2) + 2 \sum_{i=2}^{j} (j + 1 - i) E(R_1 R_i). \tag{3.28}$$

Computing the left-hand side for $j = 1, 2, \ldots$ with (3.26) or (3.27) yields $E(R_1)$ and $E(R_1 R_i)$ for $i = 1, 2, \ldots$, and finally the autocovariance function and the autocorrelation function of the stationary process $R_1, R_2, \ldots$. (We point out that the autocovariance function and the autocorrelation function are important quantities in the analysis of stochastic processes; cf. [45], for instance.) If the random variables $T_1, T_2, \ldots$ are iid mathematical renewal theory ([16], Chapter XI) yields a concrete formula for G_W. In particular, if the T_j have a continuous cumulative distribution function $G(\cdot)$ then $G_W(\cdot) = (1 - G(\cdot))/\mu$ and

$$H(R_{n+1} \mid R_0, \ldots, R_n) \geq \int_0^s H(V_{(s-u)}) \, \frac{1}{\mu} (1 - G(u)) \, du. \tag{3.29}$$

Proofs of the formulae mentioned above and a generalized version of (3.29) for which $G(\cdot)$ need not be continuous are given in [24], which also contains practical experiments. We mention that unless the ratio s/μ (as well as the ratio js/μ) is very small, the approximations (3.19) and (3.20) may be used to evaluate the integrals (3.26), (3.27) and (3.29).

Remark 3.5. (i) In the most general case (with not necessarily equidistributed instants s_n) the stochastic model consists of a family of distributions that belong to the auxiliary variables $V_{(u)}$ for $u \geq 0$. For any fixed u the parameters μ and σ define a two-parameter family of distributions. In combination with (3.24) this allows the coarse estimation of the conditional entropy $H(R_{n+1} \mid R_1, \ldots, R_n)$. The conditional entropy $H(Y_{n+1} \mid Y_1, \ldots, Y_n)$ clearly depends on the algorithmic postprocessing algorithm (see also Examples 3.12 and 3.13).
(ii) If $s_n = ns$ for all $n \in N$, the integrals (3.26), (3.27), (3.28) (and also (3.29) if the T_j additionally are iid) make the stochastic model more precise and comprehensive. The stochastic model then also comprises the autocovariance function of $R_1, R_2, \ldots$ and a sharper lower bound for the conditional entropy $H(R_{n+1} \mid R_1, \ldots, R_n)$.
(iii) Usually it is easier to derive a stochastic model for PTRNGs that exploit physical experiments (e.g., radioactive decay or quantum effects) than for PTRNGs that exploit electronic switchings (as in Examples 3.5 and 3.6). At least for smart cards the first type of PTRNGs is not relevant.

3.5 Algorithmic Postprocessing

In the last section our focus lay on the das random numbers $r_1, r_2, \ldots$ (or more precisely, on the underlying random variables $R_1, R_2, \ldots$). In this section we study the impact of the algorithmic postprocessing. For strong noise sources algorithmic postprocessing is not mandatory (cf. Section 3.6). Depending on the postprocessing algorithm, it may provide an additional security anchor. If the entropy per das random number is yet too low, a data-compressing postprocessing algorithm, which increases the average entropy per random bit, is an absolute must.

Example 3.7. Figure 3.3 shows a typical PTRNG design. The noise source generates single das bits $r_1, r_2, \ldots$ per time unit. In Step n bit r_n is XOR-ed to the feedback value of the LFSR, which is clocked synchronously to the digitization of the das bits. The right-most bit of the LFSR is the current internal random bit y_n.

If t denotes the number of LFSR cells for any initial state s_0 of the LFSR the mapping $(s_0, r_1 \ldots, r_n) \mapsto (y_{t+1}, \ldots, y_{t+n})$ is injective, implying

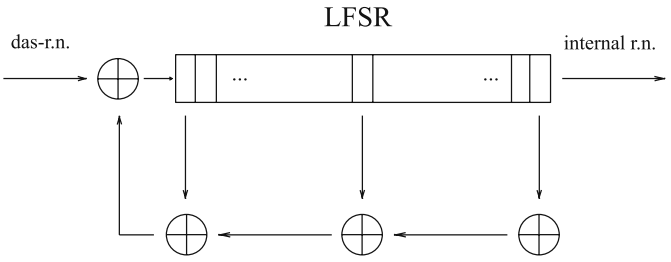

Fig. 3.3 An example of physical RNG with postprocessing.

$$H(R_1,\ldots,R_n) = H(Y_{t+1},\ldots,Y_{t+n} \mid s_0) \tag{3.30}$$

for any $n \in \mathbb{N}$. Asymptotically, this algorithmic postprocessing does not increase the average entropy per random bit, even if s_0 is unknown. It only transforms eventual weaknesses of the das random bits into others (e.g., bias into dependency). In the case of a total breakdown of the noise source the current state of the LFSR may serve as an 'entropy reserve'.

The analysis of the algorithmic postprocessing is more complicated if it shall increase the average entropy per random bit (data-compressing algorithms). In particular, its impact depends on the concrete distribution of $T_1, T_2, \ldots$.

Example 3.8. Assume that the random variables $R_1, R_2, \ldots$ are iid $B(1,p)$-distributed.
(a) Divide the sequence $R_1, R_2, \ldots$ into non-overlapping pairs $(R_1,R_2),(R_3,R_4),\ldots$ and define $Y_n := R_{2n-1} + R_{2n} \pmod 2$. If $p = 0.5 + 0.5\varepsilon$ with $\varepsilon \in [-1,1]$ elementary calculations verify that the internal random numbers are iid $B(1,p')$-distributed with $p' = 0.5 - 0.5\varepsilon^2$. For small ε we have $\log_2(0.5+0.5\varepsilon) \approx 1 - \varepsilon/\log(2)$ (linear Taylor expansion). Consequently, this algorithmic postprocessing increases the entropy per random bit from $1-\varepsilon^2/\log(2)$ to $1-\varepsilon^4/\log(2)$.
(b) (von Neumann transformation; cf. e.g., [30]) Divide the sequence $R_1, R_2, \ldots$ into non-overlapping pairs $(R_1,R_2),(R_3,R_4),\ldots$. Discard the pair (R_{2n-1},R_{2n}) if $R_{2n-1} = R_{2n}$, otherwise, the output is taken as R_{2n-1}. The internal random numbers are iid $B(1,0.5)$-distributed (ideal RNG!). In particular, $H(Y_{n+1} \mid Y_1,\ldots,Y_n) = H(Y_{n+1}) = 1$.

We point out that both postprocessings from Example 3.8 reduce the output rate of the RNG; Variant (a) by constant factor 2 while Variant (b) reduces the output by factor $p(1-p) \le 0.25$, depending on the parameter p. Both variants increase the quality of the random numbers, von Neumann's transformation even produces sequences of ideal random numbers *provided that the* $R_1, R_2, \ldots$ *are iid*. We refer the interested reader to Peres' unbiasing algorithm [34] which exploits the same idea as von Neumann's algorithm. At the cost of complexity, Peres' algorithm increases the output rate. The following example considers the impact of postprocessing Variant (a) on Markovian random variables $R_1, R_2, \ldots$ (cf. Exercises 3 and 4).

Example 3.9. We consider the algorithmic postprocessing from Example 3.8(a) but we assume that the binary-valued random variables $R_1, R_2, \ldots$ form an ergodic homogeneous Markov chain with transition matrix $P = (p_{ij})_{0\le i,j\le 1}$ (cf. Example 3.3). In particular, there exists a stationary limiting distribution ν.
We point out that the random variables $Y_1, Y_2, \ldots$ are in general no longer Markovian, which complicates the exact analysis of the entropy. If we assume that the random variables $R_1, R_2, \ldots$ are already in equilibrium state, straightforward considerations yield

$$\begin{aligned} &H(Y_{n+1} \mid Y_1,\ldots,Y_n) \ge H(Y_{n+1} \mid R_1,\ldots,R_{2n}) = H(Y_{n+1} \mid R_{2n}) = \\ &H(R_{2n+1} \oplus R_{2n+2} \mid R_{2n}) = \bar{H}(\nu(0)p_{01}(p_{10}+p_{00}) + \nu(1)p_{10}(p_{11}+p_{01})) = \\ &2p_{01}p_{10}/(p_{01}p_{10}) \ . \end{aligned} \tag{3.31}$$

On the other hand, $H(R_{2n+1} \mid R_{2n}) = \bar{H}(\nu(0)p_{01} + \nu(1)p_{11})$. We point out that the lower entropy bound from (3.31) can be further improved (cf. Exercise 3).

Example 3.10. (One-way postprocessing)
(a) A dedicated hash function $\mathrm{Ha}(\cdot)$ with output length m bits (e.g., Ha=SHA-1 with $m = 160$ or Ha=SHA-256 with $m = 256$) is applied to non-overlapping blocks $b_1, b_2, \ldots$ of consecutive $(m+k)$ das bits ($k \geq 0$). The hash values are used as internal random numbers.
(b): As in (a), but the previous internal random number (extended by k zeroes) and the current das bit block are first XOR-ed to an $(m+k)$-bit register, and then the hash function is applied to this register.

At first sight both variants may seem to be equivalent. As far as the PTRNG generates strong das random numbers this is indeed true. The second variant is yet preferable if the quality (entropy) of the das random numbers decreases (which yet should be detected by the online test!, see Section 3.6). Note that even if the PTRNG produces constant (known) sequences of das random numbers variant (b) still constitutes a strong DRNG which fulfils requirement (R3) (see Chapter 2) provided, of course, that the $(m+k)$ bit register may be viewed as uniformly distributed. This is the case if at least one 'strong' das bit block had been added to the register or several XOR-operations with low entropy blocks have been performed.

Example 3.11. (Continuation of Example 3.10)
It is practically infeasible to characterize the distribution of the internal random numbers or to calculate its entropy explicitly. On the other hand, hash functions are commonly assumed to behave in many regards as random mappings. We determine below the mean entropy for a related scenario where the algorithmic postprocessing algorithm is selected independently and uniformly from the set $\mathscr{F}_{(m+k,m)} := \{\phi\colon \{0,1\}^{m+k} \to \{0,1\}^m\}$ for each internal random number. The randomly selected algorithmic postprocessing algorithms are assumed to be publicly known. For simplicity, we assume that the das random bits are iid $B(1, 0.5)$ distributed. Consequently, the random input blocks $B_1, B_2, \ldots$ are iid uniformly distributed on $\{0,1\}^{m+k}$ for both variants. Symmetry implies

$$
\begin{aligned}
&E\left(H(\mathrm{Ha}(B_n))\right) = \qquad (3.32)\\
&-\frac{1}{|\mathscr{F}_{(m+k,m)}|} \sum_{\phi \in \mathscr{F}_{(m+k,m)}} \sum_{y \in \{0,1\}^m} \frac{|\phi^{-1}(y)|}{2^{(m+k)}} \log_2 \left(\frac{|\phi^{-1}(y)|}{2^{(m+k)}} \right) =\\
&-\frac{2^m}{|\mathscr{F}_{(m+k,m)}|} \sum_{\phi \in \mathscr{F}_{(m+k,m)}} \frac{|\phi^{-1}(y_0)|}{2^{(m+k)}} \log_2 \left(\frac{|\phi^{-1}(y_0)|}{2^{(m+k)}} \right)
\end{aligned}
$$

where $y_0 \in \{0,1\}^m$ is arbitrary but fixed. To simplify notation we temporarily use the abbreviations $v := 2^{m+k}$ and $w := 2^m$. For any fixed subset $J \subseteq \{0,1\}^{m+k}$ the probability of being the pre-image of a uniformly selected mapping $\phi \in \mathscr{F}_{(m+k,m)}$ clearly is $(w^{-1})^{|J|}(1-w)^{v-|J|}$. Hence the last term of (3.32) equals

$$-w\sum_{j=0}^{v}\binom{v}{j}\left(\frac{1}{w}\right)^{j}\left(\frac{w-1}{w}\right)^{v-j}\left(\frac{j}{v}\right)\log_2\left(\frac{j}{v}\right)= \tag{3.33}$$

$$\frac{-w}{v}\left[\sum_{j=0}^{v}\binom{v}{j}\left(\frac{1}{w}\right)^{j}\left(\frac{w-1}{w}\right)^{v-j}j\log_2(j)-\log_2(w)\sum_{j=0}^{v}\binom{v}{j}\left(\frac{1}{w}\right)^{j}\left(\frac{w-1}{w}\right)^{v-j}j\right]$$

$$=\log_2(w)-\frac{w}{v}\sum_{j=0}^{v}\binom{v}{j}\left(\frac{1}{w}\right)^{j}\left(\frac{w-1}{w}\right)^{v-j}j\log_2(j).$$

The second term quantifies the 'entropy defect' from the uniform distribution on $\{0,1\}^m$. The DeMoivre-Laplace approximation yields

$$\log_2(w)-\frac{w}{v}\sum_{j=0}^{v}\frac{\exp^{\frac{-(j-v/w)^2}{2v(1/w)(w-1)/w}}}{\sqrt{2\pi(v/w)(w-1)/w}}j\log_2(j)\approx \tag{3.34}$$

$$\log_2(w)-2^{-k}\sum_{j=0}^{v}\frac{exp^{\frac{-(j-2^k)^2}{2^{k+1}}}}{\sqrt{2\pi 2^k}}j\log_2(j)\approx$$

$$\log_2(w)-\frac{2^{-k}}{\log(2)\sqrt{2\pi 2^k}}\int_0^{\infty}\exp^{\frac{-(t-2^k)^2}{2^{k+1}}}t\log(t)\,dt.$$

(For small integer k the Poisson approximation may give better approximations.)

Note that the integrand is $\leq \exp^{\frac{-(j-2^k)^2}{2^{k+1}}}t^2$. Extending the domain of integration from $[0,\infty)$ to $(-\infty,\infty)$ provides the coarse lower entropy bound $\log_2(w)-1/((1+2^k)\log(2))$. Note that any computer algebra system enables the precise evaluation of (3.34) (Exercise 5).

Example 3.12. (Continuation of Example 3.5)
The stochastic model that was specified in (3.12) to (3.14) fits to the das random numbers from Example 3.5; see also [42], 'Justification of the stochastic model'.

The randomness of the internal random numbers follows from the uncertainty about how often the LFSR and CASR are clocked between successive outputs of internal random numbers. In contrast, if an adversary knows the number of clocks, the internal random numbers (and unpublished design parameters), he can form a system of equations to determine the internal state of LFSR and CASR, computing the next internal random numbers from the number of cycles between the next outputs.

The average increase of entropy per internal random number (k bits) is clearly limited from above by the average increase of entropy per das random number within this time period. It is yet not obvious how much entropy are really extracted by XORing k selected bits from the LFSR and CASR. In [42] an upper and a lower entropy bound for the internal random numbers is worked out. For details we refer the interested reader to [42], in particular to Theorems 1 and 2. Numerical results are collected in [42], Table 3.1. For $k=1$, $\mu/\sigma=0.01$ and time $s=60,000\cdot\mu$ between successive outputs of internal random numbers. For instance, we have $H(Y_{n+1}\mid Y_1,\ldots,Y_n)\geq 0.991$, while $H((R_{n+1(1)},R_{n+1(2)})\mid$

$(R_{1(1)},R_{1(2)}),\ldots,(R_{n(1)},R_{n(2)})) = 6.698$, providing an upper entropy bound for the internal random numbers. Note that this upper entropy bound does not provide a positive result on the question on how large k may be chosen. It yet says that in no case more than $k = 6$ bits should be extracted. For $s = 10,000 \cdot \mu$, we obtain $H(Y_{n+1} \mid Y_1,\ldots,Y_n) \geq 0.943$ for $k = 1$, (respectively, we have $H \geq 1.827$ and $H \geq 2.600$ for $k = 2$ and $k = 3$). The average entropy per das random number (two ring oscillators) then is 4.209 bit.

Example 3.13. As already pointed out, the stochastic properties of das random numbers in Example 3.6 can also be described. with the stochastic model that was specified in (3.12) to (3.14). The postprocessing is less complicated than in Example 3.12. Simplify speaking, we essentially have to replace $V_{(s-u)}$ by $V_{(s-u)} \pmod 2$ (see [24] for details).

Remark 3.6. As already pointed out in Chapter 2, Remark 2.1, for particular cryptographic applications non-complex relations exist that provide information on the unknown random numbers. For instance, N DSA- or ECDSA signatures yield a system of N linear equations in $N+1$ variables. Any disclosed ephemeral key immediately compromises the signature key x and, similarly, if an adversary knows a few bits from many ephemeral keys more sophisticated lattice-based methods also allow one to recover x (see, e.g., [18]). If the signature key x and all ephemeral keys yet were generated by an ideal RNG, these linear equations would not provide any additional information on x (neglecting other information as the public key, for instance (→ discrete log problem)); finding e.g., a DSA-signature key still would remain a 160 problem. Principally, even a small entropy defect per internal random bit may violate this 'theoretical' security property of the linear equations if N is sufficiently large (neglecting any other information on x), although it should be noted that no practial attack is known that exploits (in particular, small) entropy defects that are 'smeared' over the randomly selected ephemeral keys. If the das random numbers, as well as the internal random numbers, are iid, the von Neumann transformation (Example 3.8b) may be applied to eliminate eventual entropy defects. In the general case, it might be an option to apply an additional strong cryptographic postprocessing algorithm on the internal random numbers for applications of this type, providing a second security anchor (cf. Section 3.7).

3.6 Online Test, Tot Test, and Self Test

In Sections 3.3, 3.4, and 3.5 we discussed intensively how to estimate the entropy per random number. We introduced the concept of a stochastic model and illustrated the evaluation process by several examples. We specified and analyzed several parameter-dependent families of distributions (→ stochastic model), and we considered the impact of the algorithmic postprocessing. If a single PTRNG is concerned, the parameter(s) are estimated from random number sequences that were generated with this device. When evaluating a PTRNG design that is implemented on millions of smart cards the parameter(s) are estimated only for a few PTRNGs.

However, a concrete PTRNG in operation may generate random numbers that are much weaker than those of the carefully investigated prototypes in the laboratory. This may have several reasons, e.g., tolerances of components, ageing effects, external influences (fault attacks) or even a total failure of the noise source.

Requirement (O1) characterizes the task of a *tot test* ('tot' stands for 'total failure', not for the German adjective tot (=dead)).

- (O1) The tot test should detect a total breakdown of the noise source almost immediately.

More precisely, the tot test shall detect a total breakdown before any internal random number can be output that was influenced by das random numbers which were generated after a total failure of the noise source. (A total failure reduces the entropy of the following das random numbers to 0.) This requirement may be relaxed for memory-dependent algorithmic postprocessing algorithms since the history variables constitute an 'entropy reserve' (cf. Example 3.7) In the best case a total breakdown causes a constant sequence of das random numbers, which can easily be detected. Depending on the noise source more complicated pseudorandom patterns may occur; consider for instance the stochastic model specified in (3.12) to (3.14). The reduction of the generalized variance σ^2 to zero implies the generation of non-constant pseudorandom sequences. The tot test may be realized by a statistical test or by measurements of technical parameters such as electrical current, capacity, etc.

A *self test* should be applied after each start of the PTRNG, just to check the functionality in a qualitative sense. Self tests need not meet specific requirements.

The task of the *online test* is most critical. It shall detect any *non-tolerable* weaknesses of the random numbers while the RNG is in operation. The next section is exclusively devoted to online tests.

Besides the entropy analysis of the concrete design (inclusive the algorithmic postprocessing), the evaluator also has to verify the effectiveness of the online, tot and self tests. This also comprises the specified consequences of noise alarms, i.e., when a statistical test fails.

Remark 3.7. The literature does not consistently distinguish between online test, tot test and self test. Some authors speak of online tests or health tests, which include the tasks of the tot test and the self test. In our understanding, these tests should be distinguished at least from a logical point of view since their tasks are different. However, in concrete implementations, a single test may cover all aspects simultaneously.

3.6.1 Online Tests

The first question clearly is which random numbers should be tested. Since the external random numbers are not under the designer's control we only have the choice between the das random numbers and the internal random numbers.

Let us reconsider Example 3.7. In the worst case the noise source totally breaks down, generating a constant sequence of das bits. For a constant sequence $\ldots,0,0,\ldots$ of das random numbers the PTRNG is equivalent to a free-running LFSR which is a weak DRNG; the internal random numbers are then deterministic, having entropy 0. However, if the LFSR is not too short, the internal random numbers $y_1, y_2, \ldots$ will pass almost any collection of statistical blackbox tests unless specific characteristics of LFSR output sequences (as the linear complexity profile) are tested. Serious weaknesses of the noise source (though not a total breakdown) can hardly be detected by statistical tests. Testing the das bits would reveal at least a total breakdown immediately.

This elementary example points to a general rule, namely that usually the das bits should be tested if the RNG permits access. We point out that in very specific situations it may also be reasonable to test the internal random numbers instead. This may be the case if the das random numbers only possess low entropy and if their distribution may assume very different set parameters, e.g., if we consider a whole production series. Of course, effective tests for internal random numbers demand precise stochastic models of the internal random numbers, which seems feasible at the most for simple postprocessing algorithms (cf. Example 3.8). In contrast, Example 3.10 provides a typical counterexample. Even low entropy input causes statistically inconspicious output random numbers.

We point out that there do not exist statistical tests that are universally strong for any PTRNG design. For iid $B(1,p)$-distributed $R_1, R_2, \ldots$ (cf. Example 3.1), for instance, a monobit test which simply counts the number of zeroes and ones within a sample, is clearly appropriate. On the other hand, weaknesses of the type $\ldots,0,1,0,1,\ldots$ will not be detected by a monobit test.

Instead, statistical tests should be tailored to the stochastic model of the das random numbers or the internal random numbers (cf. Example 3.14). The distribution of $R_1, R_2, \ldots$, as well as of $Y_1, Y_2, \ldots$, and of auxiliary random variables shall remain in the class of distributions that was specified in the stochastic model under all circumstances, in particular, if non-tolerable weaknesses of the R_j occur.

Requirement (O2) to requirement (O4) formulate generic properties that online tests should fulfil.

- (O2) Non-tolerable statistical weaknesses of the das random numbers should be detected sufficiently fast.
- (O3) If the weaknesses of the random numbers are tolerable (i.e., if the 'distance' of the respective random variables from iid uniformly distributed random variables ($\rightarrow$ ideal RNG) is small) the probability for a noise alarm should be (almost) negligible.
- (O4) The online test should run fast and require only a few lines of code and little memory.

Remark 3.8. 'Real-world' PTRNGs are hardly optimal, at least not in a strict sense. However, certain deviations from an ideal RNG are tolerable. To ensure functionality, the online test should be passed with overwhelming probability if the weaknesses are tolerable ($\rightarrow$ (O3)).

Example 3.14. (i) In Example 3.1 we assumed that the das random numbers are iid $B(1,p)$-distributed. For this stochastic model, a monobit test would be appropriate.
(ii) Example 3.3 we considered a two-parameter family of distributions which may be parametrized by the transition probabilities p_{01} and p_{10}. Equality $p_{01} = p_{10} \in (0,1)$ implies $\nu = (0.5, 0.5)$. Although the random variables $R_1, R_2, \ldots$ may be far from being independent (if $p_{01} = p_{10} = 0.8$, for instance) it is very likely that the monobit test will be passed. Hence the monobit test is not effective for this PTRNG. Effective online tests should consider transition probabilities (see Exercise 6).
(iii) In Examples 3.5 and 3.12 (as well as in Examples 3.6 and 3.13) the situation is more complicated. An effective test of the internal random numbers seems to be hardly possible. Instead, the das random numbers should be tested. For time intervals with given length u, the increase of entropy essentially depends on μ and σ^2. Hence it is natural to check the arithmetic mean and the empirical variance of the number of clock cycles (as well as switchings) within time intervals of fixed length ($\rightarrow$ Exercise 7).

Statistical tests with extremely small rejection probabilities are widely spread in practice. The following example illustrates the disadvantages of those approaches. For details we refer the interested reader to [40], Section 4.

Example 3.15. The binary-valued das random numbers $r_1, \ldots, r_{320}$ are grouped into 80 non-overlapping 4-bit words which are identified with integers from 0 to 15. A χ^2-goodness-of-fit test on these 4-bit words (aka poker test) is applied

$$c := \sum_{i=0}^{15} \frac{(fr(i) - 5)^2}{5} \tag{3.35}$$

where $fr(i)$ denotes the number of i's within the eighty 4-bit words ([22], p. 69). The test is passed if the test value $c \leq 65$.

If the $R_1, R_2, \ldots$ are iid $B(1, 0.5)$-distributed the χ^2-test variable C is multinomially distributed. *If the sample size tends to infinity* the distribution of C tends to the χ^2-distribution with 15 degrees of freedom, χ^2_{15}. For moderate rejection probabilities this approximation is fully appropriate. However, $\text{Prob}(X > 65.0) = 3.4 \cdot 10^{-8}$ for a χ^2_{15} distributed random variable X, while exact computations yield $\text{Prob}(C > 65.0) = 3.8 \cdot 10^{-7}$. Although the absolute error is small the relative error is not, namely

$$\frac{|\text{Prob}(T > 65.0) - \text{Prob}(X > 65.0)|}{\text{Prob}(X > 65.0)} \approx 10.1 \ . \tag{3.36}$$

Note that for the rejection boundary 30.6, for instance, the relative error is only ≈ 0.03.

This example points to a general problem: Often, only the limit distribution of the test variable is known (i.e., when the sample size of the statistical test converges to infinity), usually only for one specific distribution (typically for iid uniformly distributed random variables). The relative approximation error at the tails of the distribution is usually considerably large. In Example 3.15 even for an ideal RNG,

10 times more online tests would fail than the designer expects if he relies upon the χ^2-approximation. This is not a security problem but may affect functionality. Other statistical tests may show the opposite behavior, i.e., less failures of the online test may occur than expected, which might induce security problems.

Another relevant question is the failure probability of the online test for random numbers that are not ideal. The limit distribution does not give an answer even for moderate rejection probabilities α (e.g., $\alpha \approx 10^{-3}$). However, moderate α allows approximations by stochastic simulations ([40], Section 7; see also Chapter 2, Section 4.3). According to a specified distribution (which is relevant for the PTRNG in evaluation), pseudorandom numbers $\widetilde{b}_1, \widetilde{b}_2, \ldots$ are generated. For example, iid $B(1, 0.485)$-distributed random variables or a Markov chain with a particular transition matrix P may be simulated. The statistical test is applied to the pseudorandom numbers in place of the das random numbers, as well as of the internal random numbers. Repeating this process many times gives an empirical cumulative distribution which provides estimates for particular rejection probabilities under the specified distribution (e.g., $\text{Prob}(C > 30.6)$ in Example 3.15 for iid $B(1, 0.485)$-distributed random variables). To obtain reliable estimates for the unknown rejection probabilities, as a rule of thumb we recommend to repeat the simulations at least $M \geq 100/\alpha$ times where α denotes the true but unknown rejection probability of interest. (Somewhat more than 100 failures of the basic test indicate that this number M should be reached.) Extremely small rejection probabilities as 10^{-9}, for instance, require gigantic workload which is a further argument to consider moderate rejection probabilities.

The key idea is to apply an elementary statistical test ($\rightarrow$ (O4)) to the random numbers but to exploit the test value by different rules which cover the requirements formulated in (O1) to (O3) above. We sketch the procedure that was introduced in Ref. [40]. We recommend the interested reader to study this reference.

At first the designer selects a statistical test, the so-called *basic test*, with regard to the specific PTRNG (as well as with regard to its stochastic model). This might be a monobit test ($\rightarrow$ Example 3.1), a test that considers one-step transition frequencies ($\rightarrow$ Example 3.3) or a χ^2 goodness of fit test, for example, provided that the required memory, the lines of code and the execution time are acceptable for the device used and the intended applications. Note, however, that the particular choice of the basic test does not affect the general principle of the online test procedure.

A *test suite* consists of at the most N basic tests. The basic test values are denoted with $c_1, c_2, \ldots$ while $hc_0, hc_1, hc_2, \ldots$ denote the *history variables*. We start with $hc_0 := \text{E}_{0,r}(C)$, the expectation of the basic test for ideal RNGs, rounded to a multiple of 2^{-c}. In Step $j \in \{1, \ldots, N\}$ a basic test is performed, and $hc_j := (1-\beta)hc_{j-1} + \beta c_j$ is computed (with $\beta = 2^{-b} \ll 1$) and rounded to a multiple of 2^{-c} where c denotes a fixed integer (typically $c \in \{5, 6\}$). Since β is a power of 2, updating the history variable only needs integer arithmetic. In Step j the following decision rules are applied:

(A): if $c_j \notin S_{(A)} \Rightarrow$ stop the test suite + noise alarm
(B): if $c_{j-k+1}, \ldots, c_j \notin S_{(B)} \Rightarrow$ stop the test suite + noise pre-alarm
(C): if $hc_j \notin S_{(C)} \Rightarrow$ stop the test suite + noise pre-alarm

If no noise alarm or noise pre-alarm occurs within the steps $1,\ldots,N$ a new test suite begins. After a noise pre-alarm a new test suite begins. Noise pre-alarms in $x \geq 1$ consecutive test suites induce a noise alarm. The designer of the PTRNG should specify the consequences of a noise alarm in the user's manual of the PTRNG (if there is any). The consequences should be adjusted to the concrete PTRNG design, the basic test, the decision rules (A) to (C), the type of application, etc. The most restrictive consequence is to stop the generation of the random numbers forever. Alternatively, the generation of further random numbers may be accepted after a specific test procedure has been passed, or a manual restart of the TRNG may be permitted. In these cases, noise alarms should be logged.

Criterion (A) covers the tot test functionality. The set $S_{(A)}$ is selected that a failure of criterion (A) is extremely unlikely for any acceptable distributions of the random numbers. Criterion (B) combines several consecutive failures of the basic test, each of them occurring with probability between 10^{-3} and 10^{-2} (to give a rule of thumb). The history variables $hc_1, hc_2, \ldots$ shall detect if the mean value of the test values $c_1, c_2, \ldots$ drifts too far from $E_{0,r}(C)$ without increasing the sample size of the basic test (for details see [40]). Note that single statistical tests may be viewed as a special case where the complements of $S_{(A)}$ and $S_{(C)}$ are empty (i.e., no condition) and $k = 1$.

In order to select appropriate sets and parameters the range of possible distributions ($\rightarrow$ stochastic model) of the das random numbers, as well as of the internal random numbers, and of auxiliary random variables, shall be divided into three subsets, possibly under consideration of the intended applications: the subset of distributions which are fully agreeable, the subset of non-tolerable distributions and the complement of these two subsets. If the distribution of the random numbers lies in the first subset a noise alarm should occur only with negligible probability, while a noise alarm should occur as soon as possible if the distribution lies in the second subset. If the true distribution is contained in the third set a noise alarm should occur sooner or later.

Example 3.16. In Example 3.1 we assumed that the das random numbers were iid $B(1,p)$-distributed. Assume that the set of tolerable and non-tolerable distributions are given by the intervals $p \in [0.49, 0.51]$ and $p \in [0.0, 0.47] \cup [0.53, 1.0]$, respectively.

If the algorithmic postprocessing from Example 3.8(a) is applied, $p \in [0.5 - \sqrt{0.02}, 0.5 + \sqrt{0.02}]$ and $p \in [0.0, 0.5 - \sqrt{0.06}] \cup [0.5 + \sqrt{0.06}, 1.0]$ define corresponding sets. For von Neumann's algorithmic postprocessing, these conditions may be even further relaxed. As long as the true distribution of the random numbers is contained in the specified family (here, if the respective random variables remain iid) only performance reasons may enforce a noise alarm. (Note that the output rate shrinks with increasing bias.)

Time intervals or events have to be specified when a basic test shall be executed, e.g., always, one basic test per second, one basic test after each external call for

random numbers, permanent testing within the idle time of the device (if the PTRNG is part of a larger cryptographic system), etc.

If the internal random numbers can be buffered it may be reasonable to apply the following test strategy: If the number of tested internal random numbers in this buffer (i.e., internal random numbers that are ready to be output) falls below a specified bound the buffer is filled up with new internal random numbers. These new internal random numbers or the das random numbers that are generated at the same time form the beginning of a sample to which the online test is applied to. When the online test has been passed also the new random numbers are ready to be output. Otherwise these numbers are deleted.

Alternatively, the online test may be applied *after* the buffer has been filled with new internal numbers. The advantage of this variant is that an adversary has absolutely no information on the stored internal random numbers. We will come back to this topic in Section 3.8 in the light of side-channel attacks and fault attacks.

With regard to the intended applications, a reasonable upper bound for the average number of noise alarms per year should be specified. Assume, for instance, that for a specific smart card ≤ 0.00002 noise alarms per year occur on average if the true distribution of the random numbers is acceptable. If the smart card is set mute after a noise alarm about 20 smart cards per million have to be exchanged unnecessarily per year.

To select appropriate parameters, the designer should be able to compute the probability for a noise alarm within a test suite. Depending on the application and on the applied test strategy ($\rightarrow$ expected number of basic tests per year) this implies the expected number of noise alarms per year. Since each basic test requires many random numbers, it is reasonable to assume that the random variables $C_1, C_2, \ldots$ are iid. Consequently, $(HC_0 = E_{0,r}(C), n_0 = 0), (HC_1, n_1), (HC_2, n_2), \ldots$ forms a homogeneous absorbing Markov chain on the finite state space $\Omega = \{j2^c \mid j2^c \in S_{(C)}\} \times \{0,1,\ldots,k-1\} \cup \{\infty\}$. The number n_j is maximum such that $c_j, c_{j-1}, \ldots, c_{j-n_j+1} \notin S_{(B)}$. The absorbing state ∞ is attained if criterion (B) or (C) is violated. (Recall that criterion (A) is violated only with negligble probability unless the PTRNG has at least (almost) totally broken down.) For details the interested reader is referred to Ref. [40], Section 6.

Example 3.17. The basic test is a χ^2 test on 128 four-bit words while $S_{(A)} = [0.0, 200.0]$, $S_{(B)} = [0.0, 26.75]$ and $S_{(C)} = [13.0, 17.0]$. Further, $\beta = 2^{-6}$, $c = 5$, $k = 3$ and $x = 3$. With regard to the application and the specified test strategy we expect 530,000 basic tests per year. The stochastic model indicates that the random numbers are iid $B(1,p)$-distributed. Table 3.1 collects some numerical results. The term p_{npa} quantifies the probability of a noise pre-alarm within a particular test suite. These figures indicate that for $|0.5 - p| > 0.025$ a noise alarm should occur soon while for $|0.5 - p| \leq 0.01$ noise alarms are relatively rare events. Note, however, that for typical smart card applications smaller noise alarm probabilities for acceptable distributions are necessary, at least if the consequence of a noise alarm is to shut the PTRNG down forever.

Table 3.1 Numerical example.

p	p_{npa}	$E\left(\frac{\#\text{ noise alarms}}{\text{year}}\right)$
0.500	0.0162	0.004
0.495	0.0184	0.006
0.490	0.0289	0.024
0.485	0.0745	0.396
0.480	0.2790	16.6
0.475	0.7470	

Remark 3.9. (see [40], Section 9)
(i) The number N should be a power of 2 to save unnecessary matrix multiplications with large matrices. This minimizes the computation time and reduces round-off errors when computing the probability p_{npa}.
(ii) The smaller $\beta := 2^{-b}$ the smaller is the influence of single basic test values on the history variables $hc_1, hc_2, \ldots$.
(iii) The history variables $HC_0, HC_1, \ldots$ may be interpreted as a 'weighted' random walk on $S_{(C)} \cap \{j2^{-c} \mid j \in Z\}$ with absorbing state ∞. The smaller c the more 'inert' is this random walk and the smaller is p_{npa}.
We recommend to choose $b, c \in \{5,6\}$. (Recall that the transition matrix P has $|\Omega|^2 = (k \cdot |S_{(C)} \cap \{j2^{-c} \mid j \in Z\}| + 1)^2$ entries.).
(iv) Although it is relevant, the meaning of the sample size m of the basic test is often neglected. Consider, for instance, a monobit test with sample size m and $S_{(B)} = [0.5m - \alpha\sqrt{m}, 0.5m + \alpha\sqrt{m}]$ for fixed α. For an ideal RNG, $\text{Prob}(C_j \in S_{(B)}) = \Phi(2\alpha) - \Phi(-2\alpha)$ regardless of m (provided that m is not extremely small). If the random numbers are iid $B(1,p)$-distributed with $p \neq 0.5$ this is yet no longer true. For $p = 0.49$, for instance, the rejection probability is almost the same as for $p = 0.5$ if m is small but almost 1 for very large m (see also Exercise 8).

The next remark addresses important aspects which have not been discussed in this chapter. Remark 3.10(i) refers to fault attacks which will be treated in Section 3.8.

Remark 3.10. (i) Primarily, online and tot tests shall detect unintended weaknesses of the noise source (ageing effects, tolerances of components, total breakdown of the noise source). However, the designer should also consider possible active attacks (fault attacks). Such attacks should either be prevented or detected by physical countermeasures, or at least the distribution of the random numbers should remain in the specified class of distributions ($\rightarrow$ stochastic model), moving to the subset of unacceptable distributions if the quality (entropy) of the random numbers goes down (cf. Section 3.8).
(ii) In this section we only considered eventual misbehavior of the analog part of the PTRNG. However, the algorithmic parts of the RNG might be implemented

incorrectly, or particular components (e.g., an LFSR or a buffer) may become defective Such failures could be detected with known-answer tests (see [20]).

3.7 Alternative Security Philosophies

In this chapter we treated stochastic models and effective online tests. As already pointed out, this shall ensure theoretical security, or more precisely, quantify the average workload to guess random numbers with a non-negligible probability if the adversary has maximum knowhow and unlimited computational power. A reliable stochastic model and effective online tests are mandatory for a successful evaluation with regard to the evaluation guidance AIS 31 ([2, 23]), which has been effective in the German certification scheme (→ Common Criteria, [8, 9]) since 2001. A large number of certification processes have verified the applicability of the AIS 31 to very different PTRNG designs. The Common Criteria themselves do not provide evaluation rules for RNGs. We point out that the AIS 20 and AIS 31 are currently updated. In particular, DRNGs and PTRNGs will be treated in a joint document.

We note that alternative security paradigms exist. The security of a PTRNG may essentially be grounded on a strong cryptographic postprocessing algorithm with memory, so that even a total breakdown of the noise source leaves a DRNG that fulfils requirements (R1) and (R2) or even (R1), (R2) and (R3). In other words, even if the entropy of the das random numbers decreases to 0 the postprocessing still guarantees computational security *provided that the entropy of the memory buffer was maximum at some instant*, which should be the case if the noise source had worked properly for some time. This reduces the requirements on the understanding of the noise source and the effectiveness of the online tests considerably. On the negative side, this does not ensure theoretical security, and (time-consuming) cryptographically strong postprocessing is mandatory. In our understanding, this construction is essentially a hybrid DRNG. The ISO standard 18031 [20] allows both alternatives, namely a (security-proofed) strong noise source with effective online tests, but also a not necessarily strong noise source with a not necessarily effective online test (aka health test) combined with a cryptographically strong postprocessing algorithm.

We note that a combination of both security paradigms, a strong noise source with effective online tests and a (possibly additional) strong cryptographic postprocessing with memory, provides two security anchors, one aiming at theoretical security, the other on practical security. A further advantage of such hybrid PTRNGs is that they may be operated in different modes, depending on the security and functional requirements of the application: as a pure PTRNG (skipping the cryptographic postprocessing), as a hybrid PTRNG (applying the cryptographic postprocessing), or as a pure/hybrid DRNG (without updating the memory of the cryptographic postprocessing algorithm continuously). The third mode might be necessary to achieve high output rates, e.g., to generate blinding or masking values (as a protection against

side-channel attacks) for high-speed encryption. Hybrid RNGs will explicitly be considered in the updated versions of the AIS 20 and AIS 31.

3.8 Side-channel Attacks and Fault Attacks

In the last decade, side-channel attacks and fault attacks have attracted enormous attention in both the scientific community and smart card industry. Unless a cryptographic device is operated in a secure environment side-channel and fault attacks constitute serious threats to any security-relevant operation. Although it is part of the overall security evaluation of the device we briefly address some aspects that concern the PTRNG, comprising the noise source, the algorithmic postprocessing algorithm and the online (tot, self) test.

Of course, the noise source (or more precisely, the das bits) should be resistant against side-channel attacks (in particular against electromagnetic radiation attacks). If the noise source is not properly shielded or sensors do not detect possible fault attacks, the duties of the online test also comprise the detection of non-tolerable weaknesses of the das random numbers (as well as of the internal random numbers) that might be induced by successful fault attacks (cf. Remark 3.10(i)). In [24] an RNG design similar to that in Example 3.6 is discussed with two noisy diodes instead of one, where the difference of the output voltages of both diodes is exploited. The basic idea is to prevent fault attacks since external influences should affect both diodes in the same way. We mention that one has to take care that the output voltages are not too different, which might lower the output rate substantially. The postprocessing algorithm should also be protected. If realized by algorithms from the cryptolibrary, effective solutions should have been developed in connection with the protection of these cryptographic algorithms anyway.

In Section 3.6 we discussed two application schemes for the online test. In the first scheme (here denoted as scheme A) the designated output data themselves are part of a sample to which the online test is applied, while in the second scheme (here denoted as scheme B) the designated output data are buffered first before the online test is applied. Assume the worst case scenario for the moment, namely, that the implementation does not prevent side-channel and/or fault attacks. For scheme A a (maximum) successful side-channel attack might reveal the tested random numbers, and a successful fault attack (applied to the before-buffered random numbers) might fool scheme B. (Note that this fault attack violated the stationarity assumption.) Vice versa, a fault attack on scheme A should cause a noise (pre-)alarm whereas in scheme B a side-channel attack on the online test does not reveal the buffered data.

If this is necessary for the concrete device and the intended conditions of use one might consider a combination of both schemes, hoping that external manipulations cannot be switched on and off at very short intervals. In the easiest case, the das random numbers and the internal random numbers are iid. Then the random numbers may be transmitted alternately over two separate lines, applying scheme A on line 1 and scheme B on line 2. The designated output values are stored in intermediate

buffers. If both online tests have been passed, the content of the intermediate buffers are XOR-ed.

3.9 Exercises

1. Consider Example 3.3 with $m = 2$ (binary-valued random variables) and transition matrix $P = (p_{ij})_{0 \le i,j \le 1}$.
   (i) Determine $H(R_{n+1} \mid R_n)$ if the RNG is in equilibrium state. In particular, compute the numerical values for the special cases $(p_{01} = p_{10} = 0.5)$, $(p_{01} = p_{10} = 0.6)$, $(p_{01} = 0.6,\ p_{10} = 0.4)$ and $(p_{01} = 0.45, p_{10} = 0.51)$.
   (ii) Compute $H(R_{n+1} \mid R_n = i)$ and $H_\infty(R_{n+1} \mid R_n = i)$ for the parameter values from (i) for $i = 0, 1$.
   (iii) Compare and discuss the results from (i) and (ii).
2. Consider a high-frequency oscillator that provides the D-input of a flip-flop. The flip-flop is latched by a low frequency oscillator. Formulate a stochastic model and try to analyse this model. Does this model remain valid if both oscillators are realized as ring oscillators?
3. Improve the lower entropy bound (3.31) in Example 3.9. Hint: Consider the conditional entropy $H(Y_{n+1} \mid Y_n, R_{2n-2})$.
4. Derive a lower entropy bound for Example 3.9 if the von Neumann postprocessing is applied.
5. Consider Example 3.11.
   (a) Evaluate the right-hand integral in (3.34) for $k \in \{3,4,5,6\}$, e.g., by numerical integration or with a computer algebra system. Compare the exact value with the coarse lower bound given in Example 3.11.
   (b) Try to estimate the entropy per internal random number for other distributions than iid $B(1, 0.5)$-distributed das random numbers, e.g., if the das random numbers are $B(1, p)$-distributed with arbitrary p.
6. Propose an effective online test for Example 3.14(ii). Justify your answer.
7. Propose an effective online test for Example 3.14(iii). Justify your answer.
8. Consider Remark 3.9. Determine $\text{Prob}(C_j \in S_{(B)})$ for several parameter values (p, α, m).

3.10 Projects

1. Implement a PTRNG on an FPGA. If you exploit more than one noise source, are these noise sources independent? Try to formulate, justify and analyze a stochastic model.
2. Implement the circuit from Exercise 2 with CMOS chips. Implement both oscillators as ring oscillators. The stochastic model determined in Exercise 2 contains parameters. Use your hardware implementation to determine numerical values for these parameters.

References

1. AIS 20. *Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators*. Version 1, 02.12.1999 (mandatory if a German IT security certificate is applied for; English translation). www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf
2. AIS 31. *Functionality Classes and Evaluation Methodology for Physical Random Number Generators*. Version 1, 25.09.2001 (mandatory if a German IT security certificate is applied for; English translation). www.bsi.bund.de/zertifiz/zert/interpr/ais31e.pdf
3. ANSI X9.82. *Random Number Generation* (Draft Version).
4. V. Bagini and M. Bucci. A Design of Reliable True Number Generators for Cryptographic Applications. In Ç. K. Koç and C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 1999*. Springer, Lecture Notes in Computer Science, Vol. 1717, pp. 204–218, Berlin, 1999.
5. M. Bucci, L. Germani, R. Luzzi, A. Trifiletti, and M. Varanonuovo. A High-Speed Oscillator-Based Truly Random Number Source for Cryptographic Applications, *IEEE Transactions on Computers*, 52, pp. 403–409, 2003.
6. M. Bucci and R. Lucci. Design of Testable Random Bit Generators. In J. Rao, B. Sunar editors, *Cryptographic Hardware and Embedded Systems—CHES 2005*. Springer, Lecture Notes in Computer Science, Vol. 3659, pp. 147–156 Berlin, 2005.
7. H. Bock, M. Bucci, and R. Luzzi. An Offset-Compensated Oscillator-Based Random Bit Source for Security Applications. In M. Joye, J.-J. Quisquater editors, *Cryptographic Hardware and Embedded Systems—CHES 2004*. Springer, Lecture Notes in Computer Science, Vol. 3156 pp. 268–281, Berlin, 2004.
8. *Common Criteria for Information Technology Security Evaluation*. Part 1–3; Version 3.1, Revision 1, (September 2006) and ISO 15408:1999.
9. *Common Methodology for Information Technology Security Evaluation* CEM-99/045. Part 2: Evaluation Methodology, Version 3.1, Revision 1, September 2006.
10. J.-S. Coron. On the Security of Random Sources. In H. Imai and Y. Zheng editors, *Public Key Cryptography—PKC 99*. Springer, Lecture Notes in Computer Science, Vol. 1560, pp. 29–42, Berlin, 1999.
11. J.-S. Coron and D. Naccache. An Accurate Evaluation of Maurer's Universal Test. In S. Tavares and H. Meijer editors, *Selected Areas in Cryptography—SAC '98*. Springer, Lecture Notes in Computer Science, Vol. 1556, pp. 57–71, Berlin, 1999.
12. L. Devroye. *Non-Uniform Random Variate Generation*. Springer, New York, 1986.
13. M. Dichtl. How to Predict the Output of a Hardware Random Number Generator. In C. D. Walter, Ç. K. Koç, C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 2003*, Springer, Lecture Notes in Computer Science 2779, pp. 181–188, Berlin, 2003.

14. M. Dichtl and J. Golic. High-Speed True Random Number Generation with Logic Gates Only. In P. Paillier, I. Verbauwhede editors, *Cryptographic Hardware and Embedded Systems—CHES 2007*, Springer, Lecture Notes in Computer Science 4727, pp. 45–62, Berlin, 2007.
15. W. Feller. *An Introduction to Probability Theory and Its Application*, Vol. 1, 4th Revised Printed, Wiley, New York, 1970.
16. W. Feller. *An Introduction to Probability Theory and Its Application*, Vol. 2, Wiley, New York, 1965.
17. W. Hoeffding and H. Robbins. The Central Limit Theorem for Dependent Random Variables. Duke *Mathematical Journal*, 15: 773–780, 1948.
18. N. Howgrave-Graham and N. Smart. Lattice Attacks on Digital Signature Schemes. Des. *Codes Cryptography*, 23: 283–290, 2001.
19. Intel Platform Security Division. *The Intel Random Number Generator*. Intel Corporation, 1999.
20. ISO/IEC 18031. *Random Bit Generation*. November 2005.
21. G. L. Jones. On the Markov Chain Central Limit Theorem. *Probability Surveys*, 1: 299–320, (2004).
22. G. K. Kanji. *100 Statistical Tests*. Sage Publications, London (1995).
23. W. Killmann and W. Schindler. *A Proposal for Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators*. Version 3.1 25.09.2001, mathematical-technical reference of [2] (English translation); www.bsi.bund.de/zertifiz/zert/interpr/trngk31e.pdf
24. W. Killmann and W. Schindler. A Design for a Physical RNG with Ro-bust Entropy Estimators, In E. Oswald and P. Rohatgi editors, *Cryptographic Hardware and Embedded Systems — CHES 2008*. Springer, Lecture Notes in Computer Science, Vol. 5154, pp. 146–163, Berlin, 2008.
25. D. P. Maher and R. J. Rance. Random Number Generators Founded on Signal and Information Theory. In Ç. K. Koç, C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 1999*. Springer, Lecture Notes in Computer Science, Vol. 1717, pp. 219–230, Berlin, 1999.
26. S. Mandal and S. Banerjee. An Integrated CMOS Chaos Generator. In S. Banerjee editor, 1st *Indian National Conference on Nonlinear Systems & Dynamics—NCNSD 2003*. Kharagpur (India), pp. 313–316, 2003.
27. G. Marsaglia. Diehard (Test Suite for Random Number Generators). www.stat.fsu.edu/∼geo/diehard.html
28. U. Maurer. A Universal Statistical Test for Random Bit Generators. *Journal of Cryptology*, 5 1992: 89–105.
29. A. J. Menezes, P. C. v. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton (1997).
30. J. v. Neumann. Various Techniques for Use in Connection with Random Digits. In A. H. Taub editor, *von Neumann Collected Works*, Vol. 5, Pergamon Press, London, pp. 768–770, 1963.
31. NIST. *Security Requirements for Cryptographic Modules*. FIPS PUB 140-1, 11.04.1994. www.itl.nist.gov/fipspubs/fip140-1.htm

32. NIST. *Security Requirements for Cryptographic Modules*. FIPS PUB 140-2 (25.05.2001) and Change Notice 1, 10.10.2001. csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
33. NIST. *Digital Signature Standard (DSS)*. FIPS PUB 186-2 (27.01.2000) with Change Notice 1, 5.10.2001. csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf
34. Y. Peres. Iterating von Neumann's Procedure for Extracting Random Bits. *Annals of Statistics*, 20, 590–597, 1992.
35. J. O. Pliam. *The Disparity Between the Work and the Entropy in Cryptology* 01.02.1999. eprint.iacr.org/complete/
36. J. O. Pliam. Incompatibility of Entropy and Marginal Guesswork in Brute-Force Attacks. In B. K. Roy, E. Okamoto editors, *Indocrypt 2000*, Springer, Lecture Notes in Computer Science, Vol. 2177, Berlin 2000, 67–79.
37. A. Rényi. On the Measure of Entropy and Information. *In Proc. Fourth Berkeley Symp. Math. Stat. Prob. 1* (1960), University of California Press, Berkeley (1961).
38. A. Rukhin et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800–22 with revisions dated (15.05.2001). csrc.nist.gov/rng/SP800-22b.pdf
39. W. Schindler. *Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators*. Version 2.0 02.12.1999, mathematical-technical reference of [1] (English translation); www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf
40. W. Schindler. Efficient Online Tests for True Random Number Generators. In Ç. K. Koç, D. Naccache, C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 2001*. Springer, Lecture Notes in Computer Science, Vol. 2162, pp. 103–117, Berlin, 2001.
41. W. Schindler and W. Killmann. Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications. In B. S. Kaliski Jr., Ç. K. Koç, C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 2002*, Springer, Lecture Notes in Computer Science vol. 2523, pp. 431–449, Berlin, 2003.
42. W. Schindler. A Stochastical Model and Its Analysis for a Physical Random Number Generator Presented at CHES 2002. In K. G. Paterson editor, *Cryptography and Coding—IMA 2003*, Springer, Lecture Notes in Computer Science 2898, pp. 276–289, Berlin, 2003.
43. C. Shannon. Mathematical Theory of Communication. *Bell System Technology*, vol. 27, pp. 379–423, 623–656, 1948.
44. T. Tkacik. A Hardware Random Number Generator. In B. S. Kaliski Jr., Ç. K. Koç, C. Paar editors, *Cryptographic Hardware and Embedded Systems—CHES 2002*, Springer, Lecture Notes in Computer Science, vol. 2523, pp. 450–453, Berlin, 2003.
45. A. M. Yaglom. *Correlation Theory of Stationary and Related Random Functions*, Vol. 1. Springer Series in Statistics, Springer, New York, 1987.

Chapter 4
True Random Number Generators for Cryptography

Berk Sunar

4.1 Introduction

Random numbers and randomization techniques are critical for modern-day cryptography. Random numbers are used to initialize key bits for secret- and public-key algorithms, seed pseudo-random number generators, provide challenges, nonces, padding bits, as well as initialization vectors in cryptographic primitives and protocols. For cryptographic applications it is crucial to generate pseudo-random bits which will be unpredictable to the adversary even at the exposure of partial information. The literature is filled with protocols that are built around state-of-the-art cryptographic primitives, yet fail in practice, due to a weak random number generator (cf. [1]).

In this chapter, we focus on practical TRNG designs that are suitable for manufacturing on common ASIC silicon process or to be implemented on reconfigurable logic platforms (e.g., FPGA, CPLD, etc.). Hence, esoteric designs and software TRNGs (e.g., TRNGs that use randomness in RAM or Disk access times [2]) are not discussed. Unfortunately, the literature of TRNG designs is rather scattered. Some designs appear in academic articles fragmented into a number of fields which specialize in digital design techniques, integrated circuits, and even physics. Many designs are simply patented and otherwise not published. Therefore, our survey of TRNG design will be incomplete. We survey a number of selected designs and discuss them in terms of their performance, weaknesses, scalability, and versatility.

In the remainder of this chapter we first discuss the building blocks of common TRNGs. We then present a potpouri of TRNG designs; incomplete, yet chosen to expose the diversity in design techniques. This is followed by a survey of post-processing techniques. Finally, we present several new research problems motivated by real-life needs.

Department of Electrical and Computer Engineering, Worcester Polytechnic Institute,
e-mail: sunar@wpi.edu

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_4,

© Springer Science+Business Media, LLC 2009

4.2 TRNG Building Blocks

A true random number generator (TRNG) is a device that utilizes physical processes to generate a random bit stream. Although there is a zoo of TRNGs available, the most popular and useful ones are commonly built from the following three components:

Entropy Source: Numerous TRNG designs have been proposed in the literature for collecting randomness from physical processes such as thermal and shot noise in circuits, jitter and metastability in circuits, Brownian motion, atmospheric noise, or even nuclear decay. The entropy source is perhaps the most critical component as it determines the available entropy. On the other hand, it should be clear that sources such as atmospheric noise [3] and nuclear decay are not viable except for fairly restricted applications or online distribution services. Furthermore, some sources exhibit biases which should be eliminated in the collection or postprocessing steps. Quantification of the available entropy and its exact statistical properties is another significant design task. Another issue is considering long-term effects which may cause the breakdown in the entropy source. Active monitoring techniques for detecting total breakdown are available. However, more subtle failures are difficult to detect in practice.

Harvesting Technique: The entropy source is tapped using a harvesting technique that ideally does not disturb the physical process above, yet collects as much entropy as possible. A large number of designs have been proposed to realize this step. Since blackbox analysis of TRNGs other than statistical tests and simple true randomness tests (Tot and restart[1]) are impossible, the harvesting mechanism should come with rigorous justification.

Postprocessing: Although this component is not needed in all designs, good design practice dictates the use of a postprocessor. The goal is to make the TRNG design more robust by postprocessing the output bits. A postprocessor may be applied to hide or eliminate biases and/or dependencies in the entropy source or harvesting mechanism. A secondary goal, which has gained quite a bit of importance due to active fault and side-channel attacks, is to provide resilience to environmental changes and to tampering by adversaries. A postprocessor may be as simple as a von Neumann corrector [4] or may be as complicated as an extractor function [5] or a one-way hash function such as SHA-1 [6]. Although one-way hash functions such as SHA-1 or MD5 provide a safety net when used for postprocessing, they make the analysis of the output distribution very difficult.

Finally, we would like to note that postprocessing algorithms do not merely improve the output distribution and make the design more robust but also bring a degree of flexibility into the design. For instance, postprocessing techniques with

[1] Briefly stated, Tot tests check for a total breakdown of the entropy source of an RNG usually caused due to material ageing effects or extreme fluctuations in the operating conditions. Restart tests verify generation of randomness by restarting the RNG from nearly identical operating conditions.

quantifiable properties allow trade-offs to be made between the quality of the output bits and the throughput of the TRNG.

4.3 Desirable Features

Thus far a large number of TRNG designs have been proposed. These designs vary significantly according to their entropy sources and the harvesting techniques they employ. Each design has its strengths and weaknesses. Some of these properties are related to performance and some are related to security and robustness. We summarize below some of the features we would like TRNGs to have.

- From a practical standpoint it is essential that TRNGs are built using a commonly available cheap silicon process. Moreover, it is highly desirable to implement TRNGs using purely digital design technique. This allows for easier integration with digital microprocessors, and also makes it possible to implement TRNGs on popular reconfigurable platforms (i.e., FPGAs and CPLDs).
- Compact and efficient design with high throughput per area and energy spent. Use of amplifiers or other analog components should be avoided, if possible. Analog components tend to consume more energy and make the analysis difficult. Note that, since we are not allowing analog components, we have to sample variations in the time domain (such as the design in [7] does) rather than the variations in the voltage levels. If strictly followed, this criterion also means that we should avoid complicated postprocessing schemes (e.g., SHA-1) or at least implement them in the software.
- It is desirable to have a mathematical justification of the entropy collection mechanism, with all assumptions empirically verified. The design should be sufficiently simple to allow rigorous analysis. To validate the output of TRNGs the DIEHARD [8] or NIST Test Suites [9] are commonly employed. These statistical tests are necessary but not sufficient. Recently, Schindler and Killman [10] sketched a methodology for evaluating true random number generators and outlined the pioneering standardization efforts of the BSI as described in [11]. They advocate rigorous testing of TRNGs and note that a statistical blackbox testing strategy may not be employed for this purpose. The AIS document provides clear evaluation criteria for TRNGs and also allows TRNG designers to present their own criteria.

4.4 Survey of TRNG Designs

In this section we present a survey of TRNG designs. The survery is certainly not exhaustive and there are many other interesting designs available. Considering that a large number of designs first appeared in patents and not in academic articles, it is also likely that many innovative designs are simply kept as trade secrets. In any case, we find it useful to present chosen representative designs to expose alternative TRNG construction techniques.

4.4.1 Baggini and Bucci

The early design introduced by Baggini and Bucci [12] as shown in Figure 4.1 uses a combination of analog and digital components for amplification and sampling of white noise. The design is build to resist variations in operating conditions and component behavior. Reference [12] gives an analytical model for the TRNG which captures the relationship between the maximum bit correlation to the output bit-rate and therefore claims that it is unnecessary to use statistical testing. The reference does not report any implementation results.

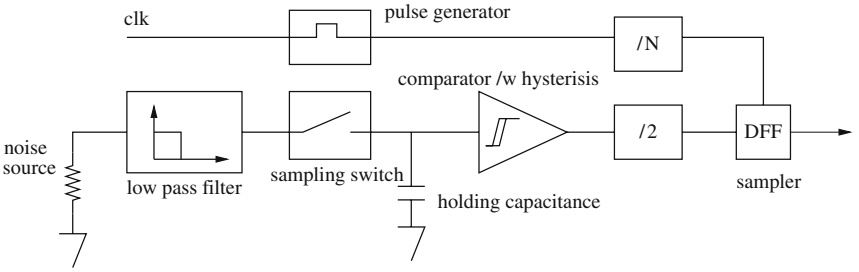

Fig. 4.1 The Baggini and Bucci TRNG Design.

4.4.2 The Intel TRNG Design

The Intel TRNG Design shown in Figure 4.2 was discussed in [6]. The entropy source of the design is thermal noise on a junction. The design uses two resistors in differential configuration to make the design more robust against power supply and environmental variations. The differential thermal noise is amplified and used to drive a voltage controlled oscillator (VCO). The VCO is then sampled by another oscillator. The output sequence is postprocessed using the von Neumann corrector and then hashed using SHA-1. As an added safety measure, the software driver that interfaces with the TRNG, implements the NIST 140-1 randomness tests monobit, runs, and poker. Jun and Kocher in [6] who have analyzed the TRNG output using 16

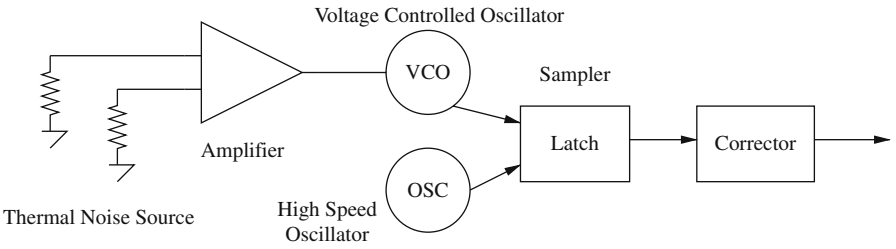

Fig. 4.2 The Intel Random Number Generator.

specialized tests and the NIST FIPS 140-1 test suite report that no weaknesses were found in the TRNG output before processing with SHA-1. The reference, however, notes that the von Neumann postprocessing technique is essential for eliminating biases in the output stream.

Since reference [6] gives only details of the security analysis we know nothing about the performance of the design, i.e., footprint, throughput, and power consumption. We may speculate that the footprint will be low due to the simplicity of the design since SHA-1 is implemented on the software side. With respect to security, we only have the blackbox analysis of Jun and Kocher. On the other hand, since the design is relatively simple, by modeling the junction noise and the oscillator jitter, one should be able to analyze the quality and performance of the TRNG output. Finally, the design has analog components, i.e., noise amplifier and voltage controlled oscillator, and therefore does not lend itself for implementation on a reconfigurable platform.

4.4.3 The Tkacik TRNG Design

The innovative design introduced in [7] randomly samples the XOR of bits chosen from a linear feedback shift register (LFSR) and a cellular automata shift register (CASR). The randomness comes from the jitter in the two free-running oscillator circuits which are used to clock the two deterministic circuits. The design is shown in Figure 4.3. The TRNG outputs 32 bits at a time. The author states that it is used with minor variations at Motorola for a number of years.

A positive aspect of the design is in its diversification. The output stream is verified using the DIEHARD [8], NIST 140-1 [9] and the Crypt-X suites [13]. The author shows that the output of the entire design has far better statistical behavior when compared to the LFSR or CASR output alone. There are no details given with regard to the performance aspects of the design. In [14] Dichtl outlines an attack on this particular TRNG construction based on two weaknesses of the design:

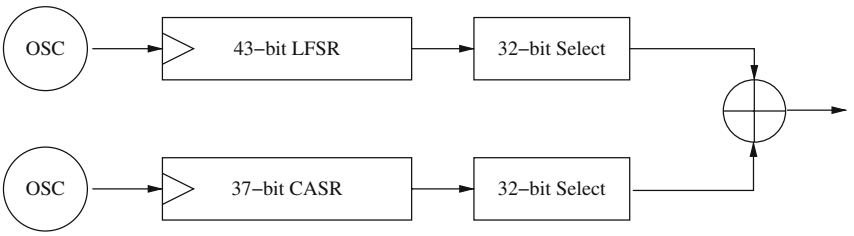

Fig. 4.3 The Tkacik TRNG Design.

- The source of entropy is fairly limited (only two oscillators are used). In fact, the LFSR and the CASR act as a pseudo-random number generator seeded with only two low-entropy oscillators.
- The design uses linear components (e.g., LFSR), and therefore the attacker can build a linear model and solve it.

The attack allows an adversary to predict the output bits assuming he/she had access to earlier bits. The treatment is theoretical and thus it is unclear if the attack would work in practice. Also, the assumption that the attacker knows some of the previously generated bits will make it impractical for many applications. On the other hand, the attack points to a dependency between output bits, and casts serious doubts about the reliability of the Tkacik TRNG. Finally, the design can be made robust by significantly lowering the output rate and/or including non-linear components. In [15] Schindler further analyzes the Tkacik design under a formulated stochastical model and develops lower and upper entropy bounds on the random output bits. Schindler also shows that the output bits carry sufficient entropy when the output is sampled 60,000 times more slowly than suggested in [7].

4.4.4 The Epstein et al. TRNG Design

In [16], a simple architecture based on bi-stable circuits is proposed. Figure 4.4 shows the basic component of the TRNG design which simply lays out many such units and computes the XOR of their output bits. A unit consists of two multiplexers and two inverters put together in a configuration that gives a metastable circuit. Note that if the select input is logic 0, then the circuit reduces to two separate single inverter oscillator rings. Alternatively, if the select input is set to logic 1, the circuit becomes functionally identical to two cascaded inverters. In the first mode, we have two free-running oscillators and in the second mode a stable circuit with no switch-

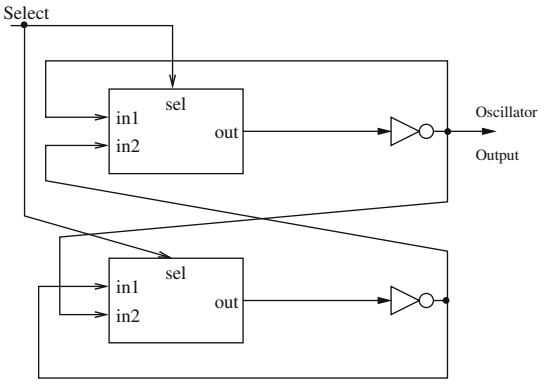

Fig. 4.4 Bi-stable memory component of the Epstein et al. TRNG design.

ing activity. Now, consider the case when the select input transitions from logic 0 to 1. Then, the two free-running oscillators may not be in the same phase and we obtain a bi-stable circuit with uncertainty in the output signal until the transitions settle.

The TRNG design which was composed of 15 instances of the components shown in Figure 4.4 and an additional 14 XOR gates was manufactured using a 0.18μ CMOS technology. All output sequences passed the DIEHARD tests after being postprocessed by the von Neumann corrector. Being constructed only from digital components, the design could be implemented on reconfigurable logic as well. Also, the design is fairly compact and should be power efficient as well. Unfortunately, reference [4] gives no information about the performance of the design. The output is verified using statistical tests. A security analysis is not provided.

4.4.5 The Fischer–Drutarovský Design

The design introduced by Fischer and Drutarovský [17] samples the jitter in a phase locked loop (PLL) on a specialized reconfigurable logic platform. The design is unique in the sense that it was the first TRNG proposal targeting FPGAs. The reference implementation targeted a particular Altera field programmable logic device family that comes with a PLL (e.g., APEX E and APEX II families) as shown in Figure 4.5. The jitter of the clock signal generated by the on-chip PLL is sampled via delay cascaded samplers organized in the configuration shown in Figure 4.6. The key idea is to use multiple samplers to be able to sample near the transition zone that is influenced by the jitter which according to [17] is of the order of only several tens of picoseconds. The multiple samples taken at regular intervals which are then XOR-ed together gives a sample from an area of the waveform that has the desired uncertainty. Finally, the output of the XOR is then downsampled using a decimator. The authors of [17] give a fairly detailed summary of the actual implementation and the design choices made. For instance, the authors note that resources need to be locked in place in the FPLD to obtain the desired routing configuration.

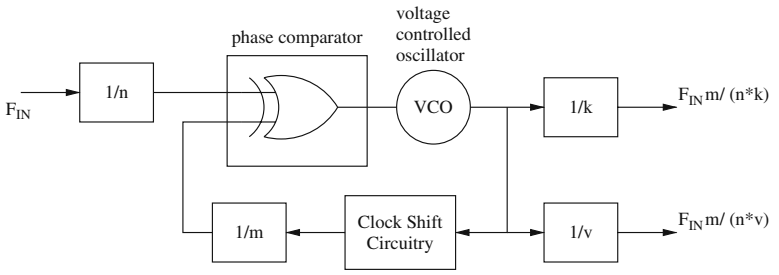

Fig. 4.5 Architecture of the programmable PLL used as the entropy source in the Fischer–Drutarovský TRNG design.

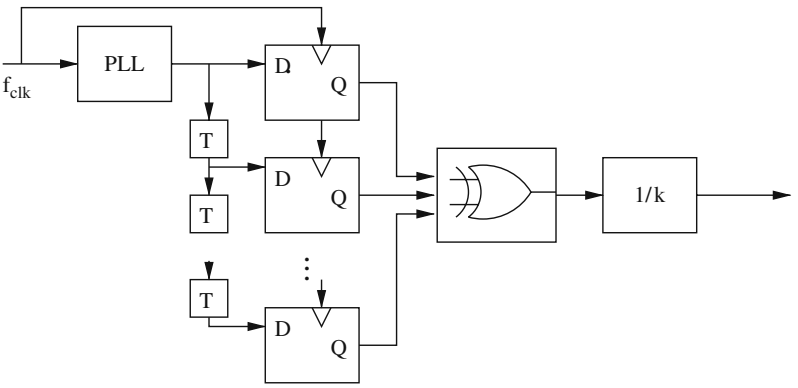

Fig. 4.6 The Fischer–Drutarovský TRNG Design.

This is especially important for the delayed samplers. The reported implementation yielded a bit-rate of nearly 70 Kbits/s. However it is not clear whether this was the upper limit of reliable operation, or whether this is merely a design choice. The generated random bit sequence was verified for statistical behavior using the NIST tests.

All in all, the Fischer–Drutarovský is important in the sense that it highlights the importance of TRNGs for reconfigurable platforms. The authors introduce the novel cascaded delayed sampler and also provide a mathematical model that allows them to pick operating points to increase the likelihood of collecting bits near the transition zones.

4.4.6 The Golić FIGARO Design

The Fibonacci oscillator [18] is shown in Figure 4.7. Basically, the structure is identical to an LFSR except for the delay elements being replaced by inverters. The feedback positions are labeled by switch values f_i. If $f_i = 1$ then the switch is closed and otherwise it is open. The switch values can be represented more conveniently in terms of the feedback polynomial which is given as follows.

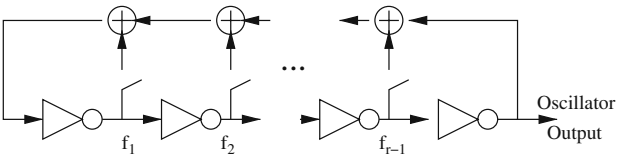

Fig. 4.7 The Fibonacci Oscillator Design.

$$f(x) = \sum_{i=0}^{r} f_i x^i \text{ where } f_0 = f_r = 1.$$

It is important that the oscillator is not stuck in a single fixed state. The necessary and sufficient conditions are given in Theorems 4.1 and 4.2.

Theorem 4.1 ([18]). *A Fibonacci ring oscillator does not have a fixed state if and only if*

$$f(x) = (1+x)h(x) \text{ and } h(1) = 1 .$$

Theorem 4.2 ([18]). *A Galois ring oscillator does not have a fixed state if and only if*

$$f(1) = 1 \text{ and } r \text{ is odd.}$$

Furthermore, for both kinds of oscillators, if $h(x)$ is chosen to be a primitive polynomial, we are guaranteed to have two cycles: a short cycle of only 2 states and a long cycle which includes the remaining $2^r - 2$ states. The Galois configuration of the oscillator ring is shown in Figure 4.8. The FIGARO (**Fi**bonacci-**Ga**lois-**R**ing-**O**scillator) TRNG design simply XORs the output of a Figaro oscillator with the output of a Galois oscillator and samples the XOR output. To eliminate local correlations and biases the author also proposes to use a self-controlled LFSR for post-processesing of the output. Later on the performance was analyzed by Dichtl and Golić (see Section 4.4.12).

4.4.7 The Kohlbrenner–Gaj Design

Similar to earlier design the Kohlbrenner–Gaj design [19] uses jitter in ring oscillators as the entropy source. What makes this design different is that, it is designed to perfectly match the CLB architecture of a Xilinx Virtex-II FPGA. The oscillator, for instance, is build into a CLB. The oscillator signal passes twice through the CLB structure and is flipped in only one of the passes (in LUT1) as shown in Figure 4.9. For clarity the clk and reset signals are not shown in the figure. The oscillation frequency is determined by the delay elements on the oscillator path, i.e., two lookup tables, four multiplexers, and two memory cells. Kohlbrenner notes that, this particular configuration gives a sufficiently stable 130 MHz oscillator signal. The TRNG samples one such oscillator with another one.

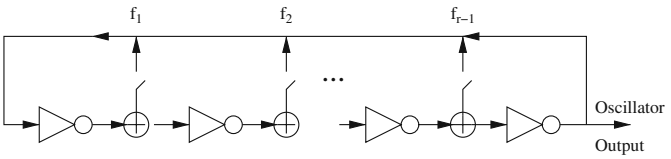

Fig. 4.8 The Galois Oscillator Design.

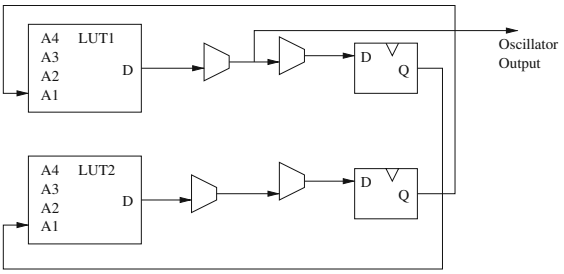

Fig. 4.9 Oscillator/CLB structure of the Kohlbrenner–Gaj Design.

The TRNG output is also postprocessed with a simple succesive XOR scheme, to eliminate biases. The reported bit-rate is of the order of several hundred kilobits/s. The exact rate depends on the strength of the XOR postprocessing scheme. Although the rate is relatively low, the design is fairly compact and its bit-rate will be sufficient in many applications. The output sequence was statistically verified using the NIST 140-1 test suite.

4.4.8 The Bucci–Luzzi Testable TRNG Design Framework

Bucci and Luzzi [20] made the observation that it is difficult, and perhaps impossible, to test the quality of TRNG outputs after complex postprocessing techniques have been employed. The authors propose to augment the designs with reset circuits that clear the state of the TRNG. This is done to support a so-called *certification mode* which establishes whether the TRNG is trustworthy. In the certification mode, the TRNG is restarted before the collection of each output bit. The objective behind the restart is to eliminate any dependencies between the collected bits. Then the output of the TRNG is either stuck in a fixed bit and no entropy is generated, or it generates independent bits. The former can be checked via a scheme that simply counts the transitions. If the transition rate is as expected, then biases in the output may be eliminated by using a stateless postprocessor. The stateless postprocessor preserves the independence among output blocks. In principle, the proposed restart approach is applicable mainly to any entropy source that permits a restart. The key point though is that the output diverges quickly from the start state into an unpredictable state. Hence, the amount of time required for an entropy source to produce diverging outputs after reset may be used as a metric.

An important side benefit of the stateless TRNG approach is that it makes detection of forcing attacks much easier when stateless linear postprocessors are used. A non-(pseudo) random bias introduced by the the attacker will be visible at the output due to the independence of the output bits and the linearity of the postprocessor.

4.4.9 The Rings Design

The rings design shown in Figure 4.10 was proposed by Sunar, Martin and Stinson in [21]. The design is very simple. Basically, free-running ring oscillator outputs are combined together via an XOR operation and then sampled. The source of randomness, is phase jitter. The main idea is to populate the output waveform with transition zones and then to sample randomly. The authors provide a mathematical framework and rigorous analysis of the quality of the output of the TRNG based on a set of assumptions at the input. Furthemore, to reduce the number of rings, the authors propose to use a resilient function for postprocessing of the TRNG output. By keeping the degree of the resilient function high, the TRNG develops a quantifiable tolerance against active adversaries. The rings design has two main contributions: the analysis framework and the introduction of resilient functions for postprocessing. The analysis builds a simple jitter model, and computes the minimum number of rings that need to be included in the design to achieve a certain fill-rate in the sampling window, at a certain confidence level. The deterministic bits collected from the unfilled portion of the sampling window are eliminated by a resilient function of appropriate strength.

An initial reference implementation of the Rings design was provided by Schellekens et al. in [22] on a Xilinx Virtex-II FPGA. The implementation produced a stream at a 2.5 Mbps bit-rate with a sampling frequency of 40 MHz and using 110 rings with 13 inverters and the resilient function constructed from the linear cyclic code $(256, 16, 113)$. The output sequence was verified using the DIEHARD and NIST tests. Schellekens et al. also observed that the Rings design is stateless and uses a linear stateless postprocessing technique (a resilient function constructed from a linear code) and therefore satisfies the criteria for testability introduced earlier by Bucci and Luzzi [20].

Finally, we should note that the Rings design received criticism in several aspects from Dichtl and Golić [23]. Among the criticisms are the independence assumption of the ring oscillators and the sampling rate. While the sampling rate may be easily reduced, it is more difficult to verify the independence of the ring oscillators when

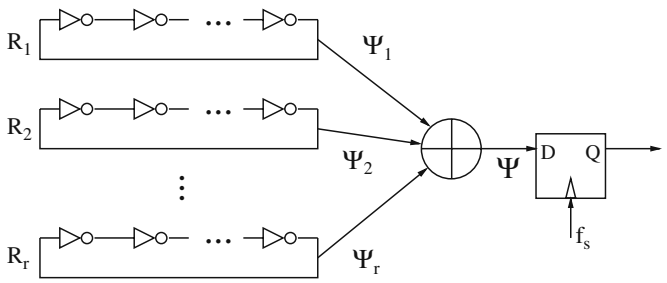

Fig. 4.10 The ring oscillators design.

a large number of rings are used. For a smaller number of rings, careful place and routing may sufficiently isolate the rings from interacting with each other. A much simpler solution is to collect only one sample from one oscillation period. In this case, ring independence is not required. It suffices to check against phase interlock which would reduce the fill-rate.

4.4.10 The PUF–RNG Design

An RNG design based on physically unclonable functions (PUFs) was proposed by O'Donnel et al. in [24]. The RNG design is build around a PUF circuit as shown in Figure 4.11. Under normal operation the output of the PUF circuit is determined by the subtle imprecisions in the delay paths created during the manufacturing process along with the challenge value supplied. Alternatively, for a particular set of challenges the delays will be closely matched and the sampling circuit will enter a meta-stable state. Hence, the output of the device will be unpredictable. While this is good news, a challenge that gives rise to metastability does so only temporarily due to temperature and voltage variations.

Hence, the PUF–RNG design *searches* for meta-stable challenges by repeatedly applying a challenge and checking if a sufficiently unstable output is obtained. Roughly stated, the same challenge is fed to the PUF circuit a fixed number of times with a fixed window length, with the hope of obtaining nearly uniform distribution at the PUF output in one window. If this is not achieved after trying a fixed number of windows, a new challenge is generated with the help of a pseudo-random number generator. When a meta-stable challenge is found, it is used to generate an output string which is further postprocessed using the von Neumann corrector.

The reference reports an implementation based on the PUF integrated into the AEGIS secure processor [25]. The PRNG, as well as the metastable challenge searching technique, is implemented in the software. The output of the RNG is verified using the NIST test suite. Unfortunately, the throughput rate is not given. The primary advantage of this design is that it makes use of an existing PUF component.

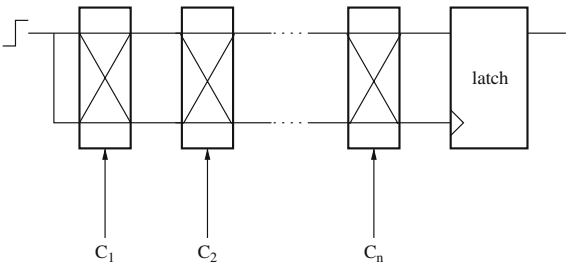

Fig. 4.11 A delay-based PUF design.

PUF circuits have become a popular tool for IC identification and for achieving tamper-resilience. Therefore, it is quite likely that a security device comes with an integrated PUF device.

4.4.11 The Yoo et al. Design

The practical aspects of the Rings design including IC routing effects, and the effects of power supply and temperature variations were investigated in [26]. The authors first note that if the signal is subsampled, then there is a chance especially at low fill-rates that the sampler may end up being stuck in an deterministic portion of the sampling window. The authors therefore recommend sampling at a frequency that is relatively prime to the oscillation frequency. The authors note that IC level effects such as phase interlock, narrow signal rejection in the XOR tree, and narrow signal attenuation affects will limit the scalability and performance of the Rings TRNG design. Furthermore, the same reference shows via experiments performed on an FPGA implementation that by changing the temperature and supply voltage, the oscillation frequency may be shifted to invalidate the relatively prime condition. Hence, the Rings TRNG may be vulnerable to non-invasive temperature and supply voltage variation attacks. Finally, to make the design robust against such attacks, the authors propose to use more than one ring length in the design. A design that features two ring lengths is proposed. The design passes the DIEHARD and NIST tests and delivers a throughput of 67 Mbps at a power consumption less than 300 mW with an area of less than 1000 LUTs. The design is also shown to be robust to temperature and power supply variations.

4.4.12 The Dichtl and Golić RNG Design

Dichtl and Golić investigated Fibonacci and Galois ring oscillators in [23]. Their analysis is primarily based on the restart technique. By restarting the oscillators from the same initial conditions they measure the time it takes to observe a bit change in the otherwise pseudo-random bitstream. Hence, the time it takes to observe a random bit determines the sampling rate and throughput of the RNG. In the same reference, the authors report an FPGA implementation of a Fibonacci ring that achieves a troughput of 6.25 Mbps.

Based on their experiments, Dichtl and Golić claim much higher entropy rates than that of traditional ring oscillators. The authors also note that the restart technique may be used as a mode of operation for the RNG and that the restart approach allows testability. Another contribution of this work is a novel two-level sampler design which reduces the bias introduced by the sampling flip-flop. With its small footprint the design seems to be ideal for embedded systems. The authors provide some preliminary justification for the performance improvement.

4.5 Postprocessing Techniques

There are several postprocessing techniques used in practice. Here we present the most popular ones.

- **Cryptographic Hash Functions:** Perhaps the most popular and most robust postprocessing technique is to run the output of a TRNG design through a cryptographically strong hash function such as SHA-1 or MD5. For instance, the Intel RNG makes use of SHA-1. From a performance perspective, implementing a full hash function for a TRNG seems like an overkill. However, from a security perspective, if properly implemented it has the important side-benefit of falling back to a pseudo-random number generator if a total breakdown occurs in the randomness source. Furthermore, the non-linearity of the hash function becomes useful if a weakness in the collection mechanism is found. A good strategy would be to implement the one-way function as the last step in software.
- **Von Neumann Corrector:** The von Neumann corrector is one of the oldest and best known postprocessing techniques and is used to eliminate localized biases. It takes pairs of bits from the random bit stream. If they are of identical value (i.e., both '0' bits or both '1' bits) it removes them from the random bit stream. If they are different, it uses one of the bits, e.g., the first bit. On average, the bit-rate will be reduced to only about 1/4 of the input bit-rate. The big advantage of the von Neumann corrector is that it is very easy to implement.
- **Extractor Functions:** The use of extractor functions was proposed by Barak, Shaltiel and Tomer in [5] with the purpose of making TRNG designs robust against changing environmental conditions. Extractor functions are powerful stateless functions with quantifiable properties originally developed as a tool for complexity theory. The authors develop a mathematicl model to capture an adversary's influence on the randomness source and give an explicit construction based on universal hash functions which is proven for its output properties even if non-local correlations exists in the input source. We give several definitions relevant to extractor functions as follows.

  Definition 4.1. The statistical distance between two distributions X and Y is defined as

  $$\varepsilon = \frac{1}{2}\sum_{a} |\mathsf{P}robX = a - \mathsf{P}robY = a| \ .$$

  In practice, we say that X is ε-close to Y and vice versa.

  Definition 4.2 (Min-Entropy). A distribution X on $\{0,1\}^n$ is said to have min-entropy k, if for all $x \in \{0,1\}^n$ $\mathsf{P}robX = x \leq 2^{-k}$.

  In general, an extractor is a function characterized with respect to its input-output behavior. An extractor is viewed as taking an input with a certain level of min-entropy k, and guarantees an output distribution that is ε close to uniform distribution. In [5] the authors provide an extension to this definition. The

authors define a function $E : \{0,1\}^n \mapsto \{0,1\}^m$ which is fixed by the choice of a public parameter. They allow an adversary to choose from 2^t distributions $D_1, D_2, \ldots, D_{2^t}$ over $\{0,1\}^n$ such that the min-entropy of each D_i is greater than k for all $i = 1, 2, \ldots, 2^t$. A public parameter π is chosen at random and independently of the choices of D_i. The adversary chooses one of the distributions, i.e., D_u. The user evaluates the extractor function using the public parameter π and a value drawn from the chosen distribution D_u. A t-resilient extractor function is defined as follows:

Definition 4.3 (t-Resilient Extractor Function, [5]). Given m, k, ε, and t, an extractor $E : \{0,1\}^n \mapsto \{0,1\}^m$ is t-resilient if with probability at the most $1 - \varepsilon$ over the choice of the public parameter π, the statistical distance of the output distribution of $E^{\pi}(X)$ to the uniform distribution is at the most ε.

In a practical setting, this means that the adversary is assumed to have control over t binary values (or 2^t internal states) of the TRNG through control of voltage, temperature, operating frequency, etc. Despite the adversary's ability, the output distribution is biased away from the uniform distribution by at the most ε. This construction gives great power to TRNG designers, since it implicitly captures *any* kind of influence by the adversary. On the other hand, from a design point of view, it is not clear how to quantify the advery's abilities and therefore it is difficult to choose design parameters for the extractor (or for the underlying universal hash function family).

- **Resilient Functions:** Resilient functions were proposed by Sunar, Martin, and Stinson in [21] as the postprocessing step for the Rings Design. The goal was to filter any deterministic bits by using the resilient function. Treating bits effected by the adversary as deterministic bits, enables one to study the tolerance properties of resilient functions against active adversaries. The reference recommends using higher resiliency degrees than necessary to remove deterministic bits. The difference between the degree of the resilient function and the number of deterministic bits expected in a sampling window quantifies the tolerance (in bits) of the TRNG to active adversaries. Resilient functions are formally defined as follows:

  Definition 4.4 (t-Resilient Function). An (n,m,t)-resilient function is a function

  $$F(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m)$$

  from $\mathbb{Z}_2^n$ to $\mathbb{Z}_2^m$ enjoying the property that, for any t coordinates $i_1, \ldots, i_t$, for any constants $a_1, \ldots, a_t$ from $\mathbb{Z}_2$ and any element y of the codomain

  $$\mathsf{P}robF(x) = y | x_{i_1} = a_1, \ldots, x_{i_l} = a_l = \frac{1}{2^m}.$$

  In the computation of this probability, all x_i for $i \notin \{i_1, \ldots, i_t\}$ are viewed as independent random variables each of which takes on the value 0 or 1 with probability 0.5.

In more informal terms, if up to any t of the input bits are deterministic and the remaining bits are random, the output of the resilient function will be perfectly random (or unpredictable). From a cryptographic viewpoint, knowledge of any t values of the input to the function does not allow one to make anything better than a random guess at the output. Resilient functions are used in a number of cryptographic applications where the adversary is assumed to have captured or determined a number of of the key bits.
A simple technique for constructing resilient functions is given in the following theorem:

Theorem 4.3. (e.g., [27]) *Let G be a generator matrix for an $[n,m,d]$ linear code C. Define a function $f : \{0,1\}^n \mapsto \{0,1\}^m$ by the rule $f(x) = xG^T$. Then f is an $(n,m,d-1)$-resilient function.*

For more information on resilient functions, and their connections to codes and designs see [28] and [29].
When compared to extractor functions, resilient functions appear to be much more limited in their capabilities of eliminating the effects of active adversaries on the output stream. The reason for this is that resilient functions are defined to work on either perfectly random or perfectly deterministic bits. In contast, extractor functions assume only a specific min-entropy at the input. On the positive side, resilient functions give perfect output distribution ($\varepsilon = 0$) and are easily constructed from codes. When linear codes are used for the construction the resilient function is also linear and therefore allows testability of the TRNG design in the sense of Bucci and Luzzi [20].

4.6 Exercises

Whenever a TRNG is to be built, several questions come to mind. Here we give an incomplete list of these questions. The reader should extend the list further by considering the context of the implementation, the platform, and development environment.

1. How small can we build it? Low footprint TRNGs are crucial for constrained applications such as RFIDs, smartcards and sensor networks. Usually only a tiny fraction of the chip area is available for the TRNG.
2. Does it scale? Trade-offs between throughput and the quality of the TRNG output are important to optimally meet application requirements at a wide variety of design points.
3. Is it robust? Robustness is an important issue especially in embedded applications, e.g., smartcards, where the user (potential attacker) has full access to the device.
4. Will we know when it fails? There is a great need for online tests. Robustness of the test circuit is also important.

References

1. I. Goldberg and D. Wagner. Randomness in the Netscape Browser. *Dr. Dobb's Journal*, January 1996.
2. D. Davis, R. Ihaka, and P. P. Fenstermacher, Cryptographic randomness from air turbulence in disk drives. In Y. Desmedt editor, *Advances in Cryptology (Crypto 94)*, vol. 839, pp. 114–120, Heidelberg, Germany: Springer-Verlag, 1994.
3. Random.org. *True random number service v2.0 beta*. www.random.org
4. J. von Neumann. *Various techniques for use in connection with random digits*, von Neumann's Collected Works, vol. 5, Pergamon, pp. 768–770, 1963.
5. B. Barak, R. Shaltiel, and E. Tomer. True Random Number Generators Secure in a Changing Environment. In Ç. K. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2003*, pp. 166–180, Berlin, Germany, Lecture Notes in Computer Science, Vol. 2779 2003. Springer-Verlag, 2003.
6. B. Jun and P. Kocher. *The Intel random number generator*, White Paper Prepared for Intel Corporation, April 1999.
7. T. E. Tkacik. A Hardware Random Number Generator In B. S. Kaliski Jr., Ç. K. Koç, C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2002*, pp. 450–453, Berlin, Germany, Lecture Notes in Computer Science, Vol. 2523. Springer-Verlag Berlin Heidelberg, 2003.
8. G. Marsaglia. *DIEHARD: A Battery of Tests of Randomness*, http://stat.fsu.edu/∼geo, 1996.
9. NIST. *A Statistical Test Suite for Random and Pseudorandom Numbers*. Special Publication 800-22, December 2000.
10. W. Schindler and W. Killmann. Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications. In B. S. Kaliski Jr., Ç. K. Koç, C. Paar, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2002*, Lecture Notes in Computer Science, Vol. 2523, pp. 431–449, Springer-Verlag Berlin Heidelberg, August 2002.
11. Anwendungshinweise und Interpretationen zum Schema (AIS). AIS 32, Version 1, *Bundesamt fr Sicherheit in der Informationstechnik*, 2001.
12. V. Bagini and M. Bucci. A Design of Reliable True Random Number Generator for Cryptographic Applications. In Ç. K. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems—CHES 1999*, pp. 204–218, Berlin, Germany, Lecture Notes in Computer Science, Vol. 1717. Springer-Verlag, 1999.
13. Crypt-X. http://www.isi.qut.edu.au/resources/cryptx/.
14. M. Dichtl. How to Predict the Output of a Hardware Random Number Generator, In C. D. Walter, Ç. K. Koç, C. Paar, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2003*, Lecture Notes in Computer Science, Vol. 2779, pp. 181–188, Springer-Verlag Berlin Heidelberg, 2003.

15. W. Schindler. A Stochastical Model and Its Analysis for a Physical Random Number Generator In K. G. Paterson editor, *Cryptography and Coding—IMA 2003*, Springer, Lecture Notes in Computer Science, vol. 2898, 276–289, Berlin, 2003.
16. M. Epstein, L. Hars, R. Krasinski, M. Rosner and H. Zheng. Design and Implementation of a True Random Number Generator Based on Digital Circuit Artifacts. In C. D. Walter, Ç. K. Koç, C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2003*, Lecture Notes in Computer Science, Vol. 2779, pp. 152–165. Springer-Verlag Berlin Heidelberg, 2003.
17. V. Fischer and M. Drutarovský. True Random Number Generator Embedded in Reconfigurable Hardware In B. S. Kaliski Jr., Ç. K. Koç, C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2002*, pp. 415–430, Berlin, Germany, Lecture Notes in Computer Science, Vol. 2523. Springer-Verlag Berlin Heidelberg, 2003.
18. J. Dj. Golić,. New methods for digital generation and postprocessing of random data. *IEEE Transactions on Computers 55(10)*: 1217–1229, 2006.
19. P. Kohlbrenner and K. Gaj. An embedded true random number generator for FPGAs International Symposium on Field Programmable Gate Arrays. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, PP. 71–78, ACM Press, New York, NY, 2004.
20. M. Bucci and R. Luzzi. Design of Testable Random Bit Generators, In J. R. Rao and B. Sunar, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2005*, Lecture Notes in Computer Science, Vol. 3659, pp. 131–146, Springer-Verlag Berlin Heidelberg, August 2005.
21. B. Sunar, W. J. Martin, and D. R. Stinson. *A Provably Secure True Random Number Generator with Built-in Tolerance to Active Attacks*, IEEE Transactions on Computers, vol. 58, no 1, p. 109–119, January 2007.
22. D. Schellekens, B. Preneel, and I. Verbauwhede FPGA Vendor Agnostic True Random Number Generator. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. pp. 1–6, August, 2006.
23. M. Dichtl and J. Dj. Golić. High-Speed True Random Number Generation with Logic Gates Only. Pascal Paillier, Ingrid verbauwhede, editors, *Proceedings of the Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop*, Vienna, Austria, September 10–13, 2007. Lecture Notes in Computer Science, vol. 4727, pp. 45-62, Springer Verlag, 2007.
24. C. W. O'Donnell, G. E. Suh, and S. Devadas. *PUF-Based Random Number Generation*. Technical Report 481, MIT CSAIL, November 2004. Available at http://www.csg.csail.mit.edu/pubs/publications.html.
25. G. E. Suh, C. W. ODonnell, I. Sachdev, and S. Devadas. *Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions*. Technical report, MIT CSAIL CSG Technical Memo 483, November 2004.

26. S.-K. Yoo, B. Sunar, D. Karakoyunlu, and B. Birand. *Practical Aspects of the Rings Design*, Available at http://ece.wpi.edu/∼sunar/preprints/rings.pdf.
27. B. Chor, O. Goldreich, J. Håstad, J. Friedman, S. Rudich, and R. Smolensky. The bit extraction problem or t-resilient functions, *26th IEEE Symposium on Foundations of Computer Science*, pp. 396–407, 1985.
28. C. J. Colbourn, J. H. Dinitz and D. R. Stinson. Applications of combinatorial designs to communications, cryptography and networking, *Surveys in Combinatorics*, 1999, pp. 37–100, (1999 British Combinatorial Conference).
29. D. R. Stinson and K. Gopalakrishnan. Applications of Designs to Cryptography, In C. D. Colbourn, and J. H. Dinitz, editors, *CRC Handbook of Combinatorial Designs*, CRC Press 1996.
30. R. A. Schulz. *Random Number Generator Circuit*. United States Patent, Patent Number 4905176, February, 27, 1990.

Chapter 5
Fast Finite Field Multiplication

Serdar Süer Erdem, Tuğrul Yanık, and Çetin Kaya Koç

5.1 Introduction

Finite fields are the most commonly used arithmetical structures in cryptography [14, 16] and coding [3, 19, 21]. Many algorithms in cryptographic and coding applications are defined in terms of finite field arithmetic operations. The elliptic curve cryptosystems [11, 17] and the Diffie-Hellman key exchange [8] algorithm are important examples of such cryptographic applications. Also, common error control codes such as Reed-Solomon and BCH codes are based on finite field theory [4, 21].

An algebraic field consists of a set and two operations defined over this set. The real numbers, the rational numbers, and the complex numbers under addition and multiplication are examples of algebraic fields. In fact, algebraic fields are the generalization of these usual number systems as described below.

- One of the field operations satisfies the general properties of the usual addition. For this operation, an identity element exists and each element has an inverse. This identity element is called additive identity or zero element.
- The other field operation satisfies the general properties of the usual multiplication. For this operation, an identity element (multiplicative identity) exists and each element, except the zero element, has an inverse. Also, this operation distributes over the first operation like the usual multiplication distributes over the usual addition.

Finite fields are algebraic fields with finite number of elements. These fields take the place of the familiar fields like the real numbers in cryptography and coding. Because they have finite number of elements, the operations on them cannot produce infinitely large results. Also, the finite field operations always produce exact results,

Gebze Institute of Technology, e-mail: serdem@gyte.edu.tr · Fatih University, Istanbul, e-mail: tyanik@fatih.edu.tr · City University of Istanbul & University of California Santa Barbara, e-mail: koc@cryptocode.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_5,

© Springer Science+Business Media, LLC 2009

not approximate results. Thus, they do not suffer from truncation errors like the floating point operations.

The fast implementation of the finite field multiplication is essential in cryptographic and coding applications. This is because the finite field addition and multiplication are the most frequently used operations in these applications. The finite field addition is relatively simple, compared to the multiplication. On the other hand, the finite field multiplication is a substantially time-consuming operation in hardware and software implementations.

In this chapter, the efficient finite field multiplication methods are discussed after giving some preliminary facts about finite fields. The discussion is handled separately for the three main classes of finite fields (prime fields, binary extension fields, and general extension fields).

5.2 Finite Fields

A finite field with q elements is denoted by $\mathbb{F}_q$. Such a field exists, if and only if $q = p^m$ for some prime p and a positive integer m. $\mathbb{F}_q$ is unique up to isomorphism. That is, every field with q elements is isomorphic to $\mathbb{F}_q$.

- $\mathbb{F}_p$ has a prime number of elements, and thus it is called prime field.
- $\mathbb{F}_{p^m}$ denotes its extension field with p^m elements.
- $\mathbb{F}_{2^m}$ is a special case of $\mathbb{F}_{p^m}$ and is called binary extension field.

The prime field $\mathbb{F}_p$ can be constructed by using integer modular arithmetic. In this construction, the field elements are represented by the set of integers $\{0,1,2,\ldots,p-1\}$. And, the field operations are defined as integer addition and multiplication modulo p.

The extension field $\mathbb{F}_{p^m}$ can be constructed by using polynomial modular arithmetic. In this construction, the field elements are represented by the polynomials over $\mathbb{F}_p$ of degree less than m. And, the field operations are defined as polynomial addition and multiplication modulo a degree m irreducible polynomial over $\mathbb{F}_p$.

The construction of the extension fields using polynomials over the prime fields is possible due to the fact that the extension field $\mathbb{F}_{p^m}$ is an m-dimensional vector space over the prime field $\mathbb{F}_p$. As an immediate result of this fact, a basis $\{\alpha_0,\alpha_1,\ldots,\alpha_{m-1}\}$ always exists in $\mathbb{F}_{p^m}$ such that each element $a \in \mathbb{F}_{p^m}$ can be given by $a = a_0\alpha_0 + a_1\alpha_1 + \cdots + a_{m-1}\alpha_{m-1}$ for a unique set of $a_i \in \mathbb{F}_p$. According to the theory of finite fields,

- A degree m irreducible polynomial over $\mathbb{F}_p$ always exists. The roots of these irreducible polynomials are in $\mathbb{F}_{p^m}$.
- Let $\alpha \in \mathbb{F}_{p^m}$ be some root of a degree m irreducible polynomial $\omega(x)$. Then, $\{1,\alpha,\alpha^2,\ldots,\alpha^{m-1}\}$ constitutes a basis for $\mathbb{F}_{p^m}$ where 1 denotes the multiplicative identity. Such a basis is called polynomial basis.

In conclusion, when α is a root of an irreducible $\omega(x)$, $\omega(x)|_{x=\alpha} = 0$ and each element of $\mathbb{F}_{p^m}$ can be given by $(a_{m-1}x^{m-1} + \cdots + a_1x + a_0)|_{x=\alpha}$ for a unique set of

$a_i \in \mathbb{F}_p$. This is why the extension field elements can be represented by the polynomials over $\mathbb{F}_p$. However, since $\omega(x) = 0$ for $x = \alpha$, all arithmetic operations are performed modulo $\omega(x)$ in this representation.

As can be understood from the discussion so far, finite field arithmetic is based on modular arithmetic, and thus requires modular reductions. Modular reduction operation is essentially computing the remainder of a division. Thus it is a costly operation unless

- A special modulus is chosen to ease the division, or
- A precomputation based on the chosen modulus is used.

The Barrett and the Montgomery algorithms are two modular reduction algorithms using precomputation. Because of the precomputation overhead, these algorithms are used if a large number of modular reductions need to be performed. Also, the Montgomery algorithm requires a domain transformation, while the Barrett algorithm does not. This domain transformation is thus a slight drawback for the Montgomery algorithm.

5.3 Multiplication in Prime Fields

The prime field $\mathbb{F}_p$ elements are represented by the set of the integers $\{0, 1, 2, \ldots, p-1\}$. Let a and b be two elements in $\mathbb{F}_p$. Let c be their product in $\mathbb{F}_p$. Then, c is defined as follows.

$$c = a \times b \bmod p\,.$$

As a result, the prime field multiplication needs two arithmetic operations:

- Integer multiplication, and
- Integer modular reduction.

The algorithms used in the modular multiplication of the integers will be studied in this section. However, the multiple precision representation, the addition, and the subtraction of the integers need to be be discussed first.

In practice, a hardware or software implementation supports a fixed w-bit word size. Each w-bit word stores an integer digit and integers are represented in the base $\beta = 2^w$. Let a be an integer in $\mathbb{F}_p$ and a_i be its ith digit. Then, the multiple precision representation for a is

$$a = (a_{n-1}, \ldots, a_2, a_1, a_0)_\beta\,.$$

Naturally, the number of digits n in this representation must satisfy $p \leq \beta^n$ so that all the integers in the set $\{0, 1, 2, \ldots, p-1\}$ can be represented.

To perform the integer addition $c = a + b$, the corresponding digits of a and b are added from the least to the most significant as follows.

$$(\varepsilon_{i+1}, c_i) = a_i + b_i + \varepsilon_i, \qquad i = 0, 1, 2, \ldots \tag{5.1}$$

Here, $\varepsilon_0 = 0$ and ε_{i+1} is the carry due to the addition of the ith digits.

Similarly, the integer subtraction $c = a - b$ is performed as follows.

$$(\varepsilon_{i+1}, c_i) = a_i - b_i - \varepsilon_i, \qquad i = 0, 1, 2, \ldots \tag{5.2}$$

Here, $\varepsilon_0 = 0$ and ε_{i+1} is the borrow due to the subtraction of the ith digits.

As seen, these digit-by-digit operations involve carry and borrow propagations which can be handled in hardware easily. Also, the general purpose processors have always the instructions "add with carry" and "subtract with borrow", which are helpful for the carry and borrow propagations.

5.3.1 Integer Multiplication

The standard way of multiplying two integers is to multiply each digit in the first by each digit in the second and combine the resulting partial products. It is easy to see that this computation requires $\mathcal{O}(n^2)$ digit operations for n-digit integers. Algorithm 1 and Algorithm 2 illustrate two different implementations of the standard integer multiplication [6, 15].

Let β be the integer base. Algorithm 1 finds $a \times b$ using the fact that

$$d = a \times b = \textstyle\sum_{i=0}^{n-1} a_i b\, \beta^i.$$

Algorithm 1 computes $a_i b$ for each a_i, then appropriately shifts and combines the results. The inner loop starting at Step 5 scans the second operand digits b_j and computes $A \times b_j = a_i \times b_j$. The result is stored into two-digit integer (H, L) in Step 6, where H and L are the higher and lower digits respectively. The previous values of the higher digit H and the running product digit d_{i+j} are also added to (H, L). Note that (H, L) can hold the result in Step 6 without any overflow because the digits $A, b_j, H, d_{i+j} \leq \beta - 1$, and thus

$$A \times b_j + H + d_{i+j} \leq (\beta - 1)(\beta - 1) + 2(\beta - 1) < \beta^2.$$

Algorithm 1: Integer multiplication (by operand scanning)

INPUT: n-digit integers a and b.
OUTPUT: $2n$-digit integer $d = a \times b$.

1. **for** $i = 0$ **to** $n - 1$ **do** $d_i = 0$
2. **for** $i = 0$ **to** $n - 1$ **do**
3. $\quad H = 0$
4. $\quad A = a_i$
5. $\quad$ **for** $j = 0$ **to** $n - 1$ **do**
6. $\quad\quad (H, L) = A \times b_j + H + d_{i+j}$
7. $\quad\quad d_{i+j} = L$
8. $\quad d_{i+n} = H$
9. **return**(d)

Algorithm 2 computes the digits of $d = a \times b$ one by one, from the least significant to the most significant. Algorithm 2 uses the fact that

$$d = \sum_{k=0}^{2n-2} \beta^k \left(\sum_{i \in I} a_i b_{k-i} \right), \qquad I = \{i \mid 0 \le i, k-i < n\}.$$

For each k, the sum $\sum_{i \in I} a_i b_{k-i}$ is computed and stored into the three-digit integer (U,H,L) in Step 6, where U and L are the most and the least significant digits respectively. Step 7 determines the kth digit of the product d as $d_k = L$. Then, Step 8 removes the digit L by shifting (U,H,L) one digit right. The remaining more significant digits U and H are used to compute the more significant digits of the product d.

Algorithm 2: Integer multiplication (by product scanning)

INPUT: n-digit integers a and b.
OUTPUT: $2n$-digit integer $d = a \times b$.

1. $(U,H,L) = (0,0,0)$
2. **for** $k = 0$ **to** $2n-2$ **do**
3. **if** $k < n$ $I = \{i \mid 0 \le i \le k\}$
4. **if** $k \ge n$ $I = \{i \mid n > i > k-n\}$
5. **for** every $i \in I$
6. $(U,H,L) \mathrel{+}= a_i \times b_{k-i}$
7. $d_k = L$
8. $(U,H,L) = (0,U,H)$
9. $d_{2n-1} = L$
10. **return**(d)

To compare the efficiencies of Algorithms 1 and 2, the inner loops of these algorithms must be considered. The inner loops of both the algorithms repeat n^2 times to perform n^2 different digit multiplications. The operations in the inner loop of Algorithm 1 are equivalent to

$$(H',L) = A \times b_j, \qquad (H,L) = (H',L) + (0,H) + (0,d_{i+j}), \qquad d_{i+j} = L.$$

These operations require four w-bit additions, two data reads (b_j, d_{i+j}), and one data write (d_{i+j}). The operations in the inner loop of Algorithm 2 are equivalent to

$$(H',L') = a_i \times b_{k-i}, \qquad (U,H,L) = (U,H,L) + (0,H',L').$$

These operations require three w-bit additions and two data reads (a_i, b_{k-i}). Also, note that the inner loops of the algorithms require multiprecision additions. These additions are performed as shown in (5.1).

Though Algorithm 1 is more straightforward to implement in hardware, Algorithm 2 is more advantageous in software. This is because Algorithm 2 requires fewer digit additions, data reads, and data writes. Here, it is assumed that the temporary variables (A, U, H, L, H', L') are held in the registers of the underlying processor; thus accessing them does not increase the data reads and writes.

5.3.2 Integer Squaring

Algorithm 3 computes the square of an integer. This algorithm is just a simplification of Algorithm 2 for the case that the multiplicands a and b are equal. Since $a = b$, the cross products satisfy $a_i b_j = a_j b_i = a_i a_j$. Thus, the number of the required digit products reduces roughly by half.

Algorithm 3 computes not all but half of the cross products using the fact

$$\sum_{\substack{i \in I \\ i \neq k-i}} a_i a_{k-i} = 2 \sum_{\substack{i \in I \\ i > k-i}} a_i a_{k-i} = 2 \sum_{\substack{i \in I \\ i < k-i}} a_i a_{k-i}$$

where $I = \{i \mid 0 \leq i, k-i < n\}$. Note that this can also be written as follows

$$\sum_{\substack{i \in I \\ i \neq k/2}} a_i a_{k-i} = 2 \sum_{\substack{i \in I \\ i > k/2}} a_i a_{k-i} = 2 \sum_{\substack{i \in I \\ i < k/2}} a_i a_{k-i} \, .$$

Algorithm 3: Integer squaring

INPUT: n-digit integer a.
OUTPUT: $2n$-digit integer $d = a^2$.

1. $(U,H,L) = (0,0,0)$
2. **for** $k = 0$ **to** $2n-2$ **do**
3. **if** $k < n$ $I = \{i \mid 0 \leq i < k/2\}$
4. **if** $k \geq n$ $I = \{i \mid n > i > k/2\}$
5. **for** every $i \in I$
6. $(U,H,L) \mathrel{+}= a_i \times a_{k-i}$
7. **if** k is even $(U,H,L) = 2(U,H,L) + a_{k/2}^2$
8. **if** k is odd $(U,H,L) = 2(U,H,L)$
9. $d_k = L$
10. $(U,H,L) = (0,U,H)$
11. $d_{2n-1} = L$
12. **return**(d)

5.3.3 Integer Modular Reduction

This section discusses the following methods for the reduction d mod p :

- The algorithms for moduli of special form
- The Barrett and the Montgomery algorithms using a precomputation based on the modulus p.

The output of the modular reduction is nothing else than the remainder of the division d/p. When the quotient calculation is omitted, the division turns into modular reduction. The multiple precision division for an arbitrary base β is a costly

operation. References [10, 15] give a good discussion of the multiple precision division and Ref. [5] presents a multiple precision modular reduction algorithm based on division.

The computation $d \bmod p$ in the base $\beta = 2$ is rather straightforward. In this case, the integer d is reduced bit by bit modulo p. Let $2^m > p \geq 2^{m-1}$ and $p \leq d = (d_{k-1}, \ldots, d_1, d_0)_2$. Then, $d_{k-1}2^{k-1} > d_{k-1}2^{k-1-m}p \geq d_{k-1}2^{k-2}$ and d can be reduced as follows.

$$d = d - d_{k-1}2^{k-1-m}p.$$

To find $d \bmod p$, the bit reductions are performed iteratively until $d < p$. Also, $d \bmod p$ can be computed by using the integer $\hat{p} = 2^m \bmod p$. Then, $d_{k-1}2^{k-1} \equiv d_{k-1}2^{k-1-m}\hat{p} \bmod p$ and d can be reduced as follows.

$$d = (d_{k-2}, \ldots, d_0)_2 + d_{k-1}2^{k-1-m}\hat{p}.$$

Algorithm 4 implements the integer modular reduction using this method.

Algorithm 4: Bit level integer modular reduction

INPUT: Integers $d = (d_{k-1}, \ldots, d_0)_2$ and $\hat{p} = 2^m \bmod p$ where $2^m > p \geq 2^{m-1}$.
OUTPUT: $d \bmod p$.

1. **while** $k > m$ **do**
2. **while** $d_{k-1} \neq 0$ **do**
3. $d = (d_{k-2}, \ldots, d_0)_2 + 2^{k-1-m}\hat{p}$
4. $k = k - 1$
5. **return**(d)

5.3.3.1 Using Special Modulus

The commonly used base to represent the integers in processors is $\beta = 2^{32}$. Thus, it is easier to perform reduction modulo a prime number which can be written as a simple sum of the powers of 2 or 2^{32}, in software and hardware implementations. The following numbers are prime and have this property,

$$\begin{aligned}
&2^{192} - 2^{64} - 1 = \beta^6 - \beta^2 - 1,\\
&2^{224} - 2^{96} + 1 = \beta^7 - \beta^3 + 1,\\
&2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 = \beta^8 - \beta^7 + \beta^6 + \beta^3 - 1,\\
&2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 = \beta^{12} - \beta^4 - \beta^3 + \beta - 1,\\
&2^{521} - 1.
\end{aligned}$$

Here, $\beta = 2^{32}$. Fast modular reduction methods can be developed for these primes [20]. Consider the prime $p = 2^{192} - 2^{64} - 1$ as an example. For $\beta = 2^{32}$,

$$\begin{array}{lll}
\beta^6 \equiv \beta^2 + 1 \mod p & \beta^8 \equiv \beta^4 + \beta^2 \mod p & \beta^{10} \equiv \beta^4 + \beta^2 + 1 \mod p\\
\beta^7 \equiv \beta^3 + \beta \mod p & \beta^9 \equiv \beta^5 + \beta^3 \mod p & \beta^{11} \equiv \beta^5 + \beta^3 + \beta \mod p.
\end{array}$$

Let $d = (d_{11}, d_{10}, d_9, d_8, d_7, d_6, d_5, d_4, d_3, d_2, d_1, d_0)_\beta$. Then, the high digits of d can be reduced efficiently as follows.

$$\begin{aligned} d_7\beta^7 + d_6\beta^6 &= (0, 0, d_7, d_6, d_7, d_6)_\beta \ , \\ d_9\beta^9 + d_8\beta^8 &= (d_9, d_8, d_9, d_8, 0, 0)_\beta \ , \\ d_{11}\beta^{11} + d_{10}\beta^{10} &= (d_{11}, d_{10}, d_{11}, d_{10}, d_{11}, d_{10})_\beta \ . \end{aligned}$$

Algorithm 5 implements this fast modular reduction method.

Algorithm 5: Integer modular reduction for $p = 2^{192} - 2^{64} - 1$

INPUT: Integer $d = (d_{11}, \ldots, d_0)_{2^{32}} < (2^{192} - 2^{64} - 1)^2$.
OUTPUT: $c = d \bmod (2^{192} - 2^{64} - 1)$.

1. Define the 6-digit integers in the base $\beta = 2^{32}$:
   $e = (d_5, d_4, d_3, d_2, d_1, d_0)_\beta$, $\quad f = (0, 0, d_7, d_6, d_7, d_6)_\beta$,
   $g = (d_9, d_8, d_9, d_8, 0, 0)_\beta$, $\quad h = (d_{11}, d_{10}, d_{11}, d_{10}, d_{11}, d_{10})_\beta$.
2. $c = e + f + g + h \bmod (2^{192} - 2^{64} - 1)$
3. **return**(c)

5.3.3.2 Barrett Modular Reduction

The Barrett method computes $c = d \bmod p$ for two integers d and p using a precomputation based on the chosen modulus [2]. The integer $c = d \bmod p$ is the remainder of the division d/p. Thus,

$$c = d - pq$$

for the quotient $q = \lfloor d/p \rfloor$. The Barrett method first finds an estimate of the quotient q using some precomputation. Let $\hat{q}$ denote this estimate. Then, the Barret method computes $c' = d - p\hat{q}$. As shown later in the text, $q - 2 \leq \hat{q} \leq q$. Thus, the Barrett method actually computes $c' = d - (q - \varepsilon)p = c + \varepsilon p$ where $\varepsilon \in \{0, 1, 2\}$. Thus, the subtraction of the modulus p from the final result one or two times may be needed for correction.

Quotient Estimation:

The Barrett method exploits the simple fact that

$$\frac{d}{p} = \left(\frac{2^k}{p}\right)\left(\frac{d}{2^{k'}}\right)\left(\frac{1}{2^{k-k'}}\right)$$

for the arbitrary integers k and k'. The divisions $2^k/p$ and $d/2^{k'}$ can be written in terms of their quotients and remainders as follows.

$$\frac{d}{p} = \left(\lfloor 2^k/p \rfloor + \frac{2^k \bmod p}{p}\right)\left(\lfloor d/2^{k'} \rfloor + \frac{d \bmod 2^{k'}}{2^{k'}}\right)\left(\frac{1}{2^{k-k'}}\right) \ .$$

Let $r^{(1)} = 2^k \bmod p$ and $r^{(2)} = d \bmod 2^{k'}$. Then, after some rearrangement,

$$\frac{d}{p} = \frac{\lfloor 2^k/p \rfloor \lfloor d/2^{k'} \rfloor}{2^{k-k'}} + \frac{\lfloor 2^k/p \rfloor r^{(2)}}{2^k} + \frac{\lfloor d/2^{k'} \rfloor r^{(1)}}{p2^{k-k'}} + \frac{r^{(1)} r^{(2)}}{p2^k} \,. \tag{5.3}$$

The Barrett algorithm estimates the quotient q of the division d/p as

$$q \approx \hat{q} = \left\lfloor \frac{\lfloor 2^k/p \rfloor \lfloor d/2^{k'} \rfloor}{2^{k-k'}} \right\rfloor, \tag{5.4}$$

i.e., the quotient of the first term in (5.3). Note that the divisions by powers of two in this estimation can be handled in software and hardware without any cost. However, the division $2^k/p$ must be precomputed for efficiency.

Estimation Error:

The quotient estimation in (5.4) will be accurate if the integers k and k' are chosen so that the last three terms in (5.3) are rational numbers less than one. In this case, the sum of the last three terms will be less than three. Let ε be the integer part of this sum. Then, $\varepsilon \leq 2$ and $q - 2 \leq \hat{q} \leq q$.

In order that the last three terms in (5.3) are less than one, the denominators must be larger than the numerators. Then,

$$\begin{aligned}
&\lfloor 2^k/p \rfloor (d \bmod 2^{k'}) \leq \lfloor 2^k/p \rfloor (2^{k'} - 1) < 2^k, \\
&\lfloor d/2^{k'} \rfloor (2^k \bmod p) \leq \lfloor d/2^{k'} \rfloor (p-1) < p2^{k-k'}, \\
&(d \bmod 2^{k'})(2^k \bmod p) \leq (2^{k'} - 1)(p-1) < p2^k.
\end{aligned}$$

The inequalities above always hold, if $2^{k'} \leq p$, $d \leq 2^k$, and $k' \leq k$. Then, for $p \leq d$, the parameters k and k' can be chosen as

$$k \geq \log_2 d, \qquad k' \leq \log_2 p.$$

Barrett Algorithm:

Algorithm 6 implements the Barrett algorithm.

Algorithm 6: Barrett modular reduction

INPUT: The integers d and p.
OUTPUT: $c = d \bmod p$.

1. Precompute $\hat{p} = \lfloor 2^k/p \rfloor$ where $k \geq \log_2 d$.
2. $u = \lfloor d/2^{k'} \rfloor$ where $k' \leq \log_2 p$.
3. $\hat{q} = \lfloor \hat{p}u/2^{k-k'} \rfloor$
4. $c = d - \hat{q}p$
5. **while**$(c \geq p)$ $\quad c = c - p$
6. **return** c

Multiprecision Implementation:

Let $d = (d_{l-1}, \ldots, d_0)_\beta$ and $p = (p_{n-1}, \ldots, p_0)_\beta$ where $p \leq d$ and β is a power of two. The integers k and k' can be chosen as

$$k = l \log_2 \beta \geq \log_2 d, \qquad k' = (n-1) \log_2 \beta \leq \log_2 p.$$

Algorithm 7 implements the Barrett algorithm for $2^k = \beta^l$ and $2^{k'} = \beta^{n-1}$.

Algorithm 7: Multiprecision Barrett modular reduction

INPUT: Integers $d = (d_{l-1}, \ldots, d_0)_\beta$ and $p = (p_{n-1}, \ldots, p_0)_\beta > \beta^{n-1}$.
OUTPUT: $c = d \bmod p$.

1. Precompute $\hat{p} = (\hat{p}_{l-n}, \ldots, \hat{p}_0)_\beta = \lfloor \beta^l / p \rfloor$.
2. $u = (u_{l-n}, \ldots, u_0)_\beta = (d_{l-1}, \ldots, d_{n-1})_\beta$
3. $v = \hat{p}u$
4. $\hat{q} = (\hat{q}_{l-n}, \ldots, \hat{q}_0)_\beta = (v_{2(l-n)+1}, \ldots, v_{l-n+1})_\beta$
5. $c = (c_n, \ldots, c_0)_\beta = d - \hat{q}p$
6. **while**$(c \geq p)$ $\quad c = c - p$
7. **return**(c)

- Step 2 computes the integer $u = \lfloor d/\beta^{n-1} \rfloor$.
- Step 3 computes the product $v = \hat{p}u$, which can be approximated as

  $$v \approx v' = \textstyle\sum_{i+j \geq l-n-1} \hat{p}_i u_j \beta^{i+j}.$$

  Note that the error due to the ignored terms is

  $$v - v' = \textstyle\sum_{0 \leq i+j \leq l-n-2} \hat{p}_i u_j \beta^{i+j} \leq \sum_{0 \leq k \leq l-n-2} (k+1)(\beta-1)^2 \beta^k.$$

  It can be shown that $v - v' \leq \beta^{l-n-1}((l-n-1)\beta - l + n) + 1$. Moreover,

  $$v - v' < \beta^{l-n+1}$$

  for $\beta \geq (l-n-1)$.
- Step 4 finds the estimate $\hat{q} = \lfloor v/\beta^{l-n+1} \rfloor$. $\hat{q}$ can be approximated as $\lfloor v'/\beta^{l-n+1} \rfloor$. The resulting error will be less than one as shown below.

  $$\lfloor v/\beta^{l-n+1} \rfloor - \lfloor v'/\beta^{l-n+1} \rfloor \leq 1\,.$$

- Step 5 finds $(d \bmod p + \varepsilon p)$ where $\varepsilon = q - \hat{q}$. Since $p < \beta^n$ and ε is a small number, the result of Step 5 will not be more than n digits in the worst case. Thus, only the lower n digits of the product $\hat{q}p$ need to be computed in this step. Step 6 removes εp.

5.3.3.3 Montgomery Modular Reduction

The Montgomery modular reduction computes $d\theta^{-1} \bmod p$ for two integers d and p [13, 18]. Here, θ is preferably a power of two such that $\gcd(p,\theta) = 1$ and $p\theta > d$. The Montgomery method requires some precomputation and domain transformation.

The Montgomery method is used to reduce the products of the integers represented in the Montgomery residue domain. Let a' and b' be two integers. Let $c' = a'b' \bmod p$ be their modular product. In the Montgomery residue domain, these integers are represented by

$$a = a'\theta \bmod p\,, \qquad b = b'\theta \bmod p\,, \qquad c = c'\theta \bmod p\,.$$

Let $d = ab$ be the product of the integers in the residue domain. Then, the Montgomery modular reduction $d\theta^{-1} \bmod p$ yields their product in the residue domain c as shown below.

$$\begin{aligned} d\theta^{-1} \bmod p &= (a'\theta \bmod p)(b'\theta \bmod p)\theta^{-1} \bmod p \\ &= a'b'\theta \bmod p \\ &= c'\theta \bmod p \\ &= c. \end{aligned}$$

The Montgomery method computes $c = d\theta^{-1} \bmod p$ as follows.

$$c = \frac{d - (dp^{-1} \bmod \theta)p}{\theta} - \varepsilon p \tag{5.5}$$

where $d < p\theta$ and $\varepsilon \in \{0,1\}$. This computation leads to an efficient modular reduction algorithm when θ is a power of two and $p^{-1} \bmod \theta$ is precomputed.

The correctness of the Montgomery modular reduction method can be shown by using the Bezout's identity. Because θ and p are relatively prime,

$$\theta\hat{\theta} + p\hat{p} = \gcd(\theta, p) = 1$$

where $\hat{\theta} = \theta^{-1} \bmod p$ and $\hat{p} = p^{-1} \bmod \theta$. Then, $d = d\theta\hat{\theta} + dp\hat{p}$. Since $d < p\theta$,

$$\begin{aligned} d &= d\theta\hat{\theta} + dp\hat{p} \bmod p\theta \\ &= (d\theta\hat{\theta} \bmod p\theta) + (dp\hat{p} \bmod p\theta) + \varepsilon p\theta \end{aligned}$$

where $\varepsilon \in \{0,1\}$. Moreover, it can be written that

$$\begin{aligned} d &= (d\hat{\theta} \bmod p)\theta + (d\hat{p} \bmod \theta)p + \varepsilon p\theta \\ &= (d\theta^{-1} \bmod p)\theta + (dp^{-1} \bmod \theta)p + \varepsilon p\theta \\ &= c\theta + (dp^{-1} \bmod \theta)p + \varepsilon p\theta \end{aligned}$$

using the rules of modular arithmetic. See that Equation (5.5) can be obtained by rearranging the above equation.

Montgomery Algorithm and its Multiprecision Implementation:

Let $d = (d_{l-1}, \ldots, d_0)_\beta$ and $p = (p_{n-1}, \ldots, p_0)_\beta$ where β is a power of two. The integer θ can be chosen as $\theta = \beta^{l-n}$. Then, Equation (5.5) is given by

$$c = \left\lfloor \frac{d}{\beta^{l-n}} \right\rfloor - \left\lfloor \frac{(d\hat{p} \bmod \beta^{l-n})p}{\beta^{l-n}} \right\rfloor - \varepsilon p$$

where $\gcd(\beta, p) = 1$, $\hat{p} = p^{-1} \bmod \beta^{l-n}$, and $d < p\beta^{l-n}$.

Algorithm 8 implements the Montgomery algorithm.

Algorithm 8: Multiprecision Montgomery modular reduction

INPUT: Integers $d = (d_{l-1}, \ldots, d_0)_\beta$ and $p = (p_{n-1}, \ldots, p_0)_\beta$ such that $\gcd(\beta, p) = 1$ and $d < p\beta^{l-n}$.
OUTPUT: $c = d\beta^{-(l-n)} \bmod p$.

1. Precompute $\hat{p} = (\hat{p}_{l-n-1}, \ldots, \hat{p}_0)_\beta = p^{-1} \bmod \beta^{l-n}$.
2. $u = (u_{l-n-1}, \ldots, u_0)_\beta = d\hat{p} \bmod \beta^{l-n}$
3. $v = up$
4. $c = (d_{l-1}, \ldots, d_{l-n})_\beta - (v_{l-1}, \ldots, v_{l-n})_\beta$
5. **while**$(c \geq p)$ $\quad c = c - p$
6. **return**(c)

- Step 2 computes the $(l-n)$-digit integer $u = d\hat{p} \bmod \beta^{l-n}$ as follows.

$$\begin{aligned} u &= (\textstyle\sum_{i+j<l-n} d_i \hat{p}_j \beta^{i+j}) \bmod \beta^{l-n} \\ &= \textstyle\sum_{i+j<l-n-1} d_i \hat{p}_j \beta^{i+j} + (\sum_{i+j=l-n-1} d_i \hat{p}_j \bmod \beta)\beta^{l-n-1}. \end{aligned}$$

- Step 3 computes the product $v = up$, which can be approximated as

$$v \approx v' = (\textstyle\sum_{l-n-2 \leq i+j} u_i p_j \beta^{i+j}).$$

Note that the error due to the ignored terms is

$$v - v' = \textstyle\sum_{0 \leq i+j \leq l-n-3} \hat{p}_i u_j \beta^{i+j} \leq \sum_{0 \leq k \leq l-n-3} (k+1)(\beta-1)^2 \beta^k.$$

It can be shown that $v - v' \leq \beta^{l-n-2}((l-n-2)\beta - l + n + 1) + 1$. Moreover,

$$v - v' < \beta^{l-n}$$

for $\beta \geq (l-n-2)$.

- Step 4 computes

$$c = \lfloor d/\beta^{l-n} \rfloor - \lfloor v/\beta^{l-n} \rfloor \approx \lfloor d/\beta^{l-n} \rfloor - \lfloor v'/\beta^{l-n} \rfloor .$$

Note that, when v is approximated by v', the error in c is less than one since $v - v' < \beta^{l-n}$.

- After Step 4, $c = d \bmod p + \varepsilon p$ for a small number ε. Step 5 removes εp.

5.4 Multiplication in Binary Extension Fields

The binary extension field $\mathbb{F}_{2^m}$ elements are represented by the set of the polynomials of degree less than m with coefficients in $\mathbb{F}_2$. That is

$$\mathbb{F}_{2^m} = \{a(x) \mid a(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0,\ a_i \in \mathbb{F}_2\}.$$

Let $a(x)$ and $b(x)$ be two elements in $\mathbb{F}_{2^m}$. Let $c(x)$ be their product in $\mathbb{F}_{2^m}$. Then, $c(x)$ is defined as follows.

$$c(x) = a(x) \times b(x) \bmod \omega(x)$$

where $\omega(x)$ is a degree m irreducible polynomial over $\mathbb{F}_2$. As a result, the binary extension field multiplication needs two arithmetic operations:

- Polynomial multiplication over $\mathbb{F}_2$, and
- Polynomial modular reduction over $\mathbb{F}_2$.

The algorithms used in the modular multiplication of the polynomials over $\mathbb{F}_2$ will be studied in this section. However, the multiple precision representation, the addition, and the subtraction of the polynomials over $\mathbb{F}_2$ need to be discussed first.

Let a fixed w-bit word size be supported in a hardware or software implementation. Each w-bit word can store w polynomial coefficients since the polynomial coefficients are in $\mathbb{F}_2$ and represented by the integers $\{0, 1\}$. Let $a(x)$ be a polynomial over $\mathbb{F}_2$ and a_i be its ith coefficient. Let $A_i = \sum_{k=0}^{w-1} a_{iw+k}x^k$. Then, each A_i is a w-coefficient polynomial stored in a single word and the multiple precision representation for $a(x)$ is

$$a(x) = A_{n-1}x^{(n-1)w} + \ldots + A_2 x^{2w} + A_1 x^w + A_0 . \tag{5.6}$$

In this representation, n is the number of the single word polynomials A_i. Naturally, n must satisfy the inequality $x^m \leq x^{wn}$ so that all the polynomials of degree less than m over $\mathbb{F}_2$ can be represented.

To perform the polynomial addition $c(x) = a(x) + b(x)$ and the polynomial subtraction $c(x) = a(x) - b(x)$, the corresponding coefficients of $a(x)$ and $b(x)$ must be added and subtracted in $\mathbb{F}_2$ respectively. The binary-valued elements of $\mathbb{F}_2$ $\{0,1\}$ are added or subtracted modulo 2. As a result, the addition and the subtraction in $\mathbb{F}_2$ are just equivalent to XOR operation. Thus, $c(x) = a(x) \pm b(x)$ are performed by bitwise XORing the corresponding words as follows.

$$C_i = A_i \text{ XOR } B_i, \qquad i = 0, 1, 2, \ldots \tag{5.7}$$

The bitwise XOR operation is ubiquitously found in hardware and software implementations.

5.4.1 Polynomial Multiplication over $\mathbb{F}_2$

Let $d(x) = a(x)b(x)$. If $a(x)$ and $b(x)$ are represented as shown in (5.6),

$$\begin{aligned} d(x) &= (\textstyle\sum_{i=0}^{n-1} A_i x^{iw}) b(x) \\ &= \textstyle\sum_{i=0}^{n-1} (\sum_{k=0}^{w-1} a_{iw+k} x^k) x^{iw} b(x) \\ &= \textstyle\sum_{k=0}^{w-1} x^k \sum_{i=0}^{n-1} a_{iw+k} x^{iw} (\sum_{j=0}^{n-1} B_j x^{jw}) \\ &= \textstyle\sum_{k=0}^{w-1} x^k \sum_{i=0}^{n-1} a_{iw+k} \sum_{j=0}^{n-1} B_j x^{(i+j)w}. \end{aligned}$$

This discrete summation formula leads to the right-to-left and the left-to-right multiplication methods implemented in Algorithms 9 and 10 for the polynomials over $\mathbb{F}_2$, respectively.

Algorithm 9: Right-to-left comb method

INPUT: Polynomials over $\mathbb{F}_2$ $a(x)$ and $b(x)$ of degree less than $m \leq nw$.
OUTPUT: $d(x) = a(x)b(x)$.

1. **for** $i = 0$ **to** $2n-1$ **do** $D_i = 0$
2. **for** $k = 0$ **to** $w-1$
3. **for** $i = 0$ **to** $n-1$
4. **if** the kth bit of A_i is 1
5. **for** $j = 0$ **to** n
6. $D_{i+j} = D_{i+j} + B_j$
7. **if** $k \neq w-1$ **then** $b(x) = \sum_{l=0}^{n} B_l x^{lw} = xb(x)$
8. **return**$(d(x))$

Algorithm 10: Left-to-right comb method

INPUT: Polynomials over $\mathbb{F}_2$ $a(x)$ and $b(x)$ of degree less than $m \leq nw$.
OUTPUT: $d(x) = a(x)b(x)$.

1. **for** $i = 0$ **to** $2n-1$ **do** $D_i = 0$
2. **for** $k = w-1$ **downto** 0
3. **for** $i = 0$ **to** $n-1$
4. **if** the kth bit of A_i is 1
5. **for** $j = 0$ **to** $n-1$
6. $D_{i+j} = D_{i+j} + B_j$
7. **if** $k \neq 0$ **then** $d(x) = \sum_{l=0}^{2n-1} D_l x^{lw} = xd(x)$
8. **return**$(d(x))$

Algorithm 11 is a faster implementation of the left-to-right comb method given in Algorithm 10. However, this implementation requires more memory. Algorithm 11 computes all the possible products $b(x)u(x)$ where $u(x)$ is a polynomial of degree less than four and stores the resulting polynomials into the local variable space as a lookup table.

Algorithm 11: Left-to-right comb method with 4-bit window

INPUT: Polynomials over $\mathbb{F}_2$ $a(x)$ and $b(x)$ of degree less than $m \leq nw-3$.
OUTPUT: $d(x) = a(x)b(x)$.

1. Compute $f(u(x)) = b(x)u(x)$ for all $u(x)$ with $\deg(u(x)) < 4$.
2. **for** $i = 0$ **to** $2n-1$ **do** $D_i = 0$
3. **for** $k = 4\lfloor (w-1)/4 \rfloor$ **downto** 0 **by** 4
4. **for** $i = 0$ **to** $n-1$
5. $u(x) = \lfloor A_i / x^k \rfloor \bmod x^4$
6. $b'(x) = \sum_{l=0}^{n-1} B'_l x^{lw} = f(u(x))$
7. **for** $j = 0$ **to** $n-1$
8. $D_{i+j} = D_{i+j} + B'_j$
9. **if** $k \neq 0$ **then** $d(x) = \sum_{l=0}^{2n-1} D_l x^{lw} = x^4 d(x)$
10. **return**$(d(x))$

Note that the polynomial $u(x)$ has $2^4 = 16$ different possible values and the product $b(x)u(x)$ has $m+3$ coefficients. Thus, the required memory space for the lookup table is $16(m+3)$ bits. Algorithm 11 multiplies each four consecutive polynomial terms of $a(x)$ by $b(x)$ using the lookup table. The window size four can be increased, but then a larger lookup table will be needed and the overhead of the lookup table computation will increase.

5.4.2 Polynomial Squaring over $\mathbb{F}_2$

Let $a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0$ be a polynomial over $\mathbb{F}_2$. The square of a polynomial is given by

$$a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} + \sum_{0 \le j < i < m} \underbrace{2a_{i+j}x^{i+j}}_{0} .$$

As shown above, multiplication by 2 yields zero result in a characteristic 2 field. Thus, the cross products are zero and

$$a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} = a_{m-1}x^{2(m-1)} + \cdots + a_1x^2 + a_0 .$$

Algorithm 12 computes the square of the polynomials over $\mathbb{F}_2$ where the word size w is divisible by 8. This algorithm precomputes and stores the square of all possible polynomials of degree less than 8 in a lookup table. Then, it computes the square of each consecutive eight terms of the input polynomial using the lookup table. The lookup table contains 2^8 polynomials of degree less than 16, and thus is of size 512 bytes.

Algorithm 12: Squaring of Polynomials over $\mathbb{F}_2$ where $8 \mid w$

INPUT: A polynomial over $\mathbb{F}_2$ $a(x)$ of degree less than $m \le nw$.
OUTPUT: $d(x) = a(x)^2$.

1. Precompute $f(u(x)) = u(x)^2$ for all $u(x)$ with $\deg(u(x)) < 8$.
2. **for** $i = 0$ **to** $n-1$
3. $\quad C_{2i} = 0$
4. $\quad$ **for** $k = 8\lfloor (w/2-1)/8 \rfloor$ **downto** 0 **by** 8
5. $\quad\quad u(x) = \lfloor A_i/x^k \rfloor \bmod x^8$
6. $\quad\quad C_{2i} = C_{2i} + f(u(x))x^{2k}$
7. $\quad C_{2i+1} = 0$
8. $\quad$ **for** $k = 8\lfloor (w-1)/8 \rfloor$ **downto** $8\lfloor w/16 \rfloor$ **by** 8
9. $\quad\quad u(x) = \lfloor A_i/x^k \rfloor \bmod x^8$
10. $\quad\quad C_{2i+1} = C_{2i+1} + f(u(x))x^{2k}$
11. **return**$(d(x))$

5.4.3 Polynomial Modular Reduction over $\mathbb{F}_2$

This section discusses the following methods for the modular reduction of the polynomials $d(x) \bmod \omega(x)$:

- The algorithms for moduli of special form
- The Barrett and the Montgomery algorithms using a precomputation based on the modulus $\omega(x)$.

In general, $d(x) \bmod \omega(x)$ over $\mathbb{F}_2$ can be performed as follows.

$$d(x) = d(x) + d_k x^{k-m} \omega(x)$$

where $k = \deg(d(x))$ and $m = \deg(\omega(x))$. To find $d(x) \bmod \omega(x)$, the coefficient reductions are performed iteratively until $d(x) < \omega(x)$. Algorithm 13 implements the integer modular reduction using this method.

Algorithm 13: Polynomial modular reduction over $\mathbb{F}_2$

INPUT: Polynomials $d(x)$ and $\omega(x)$ where $\deg(d(x)) = k$ and $\deg(\omega(x)) = m$.
OUTPUT: $d(x) \bmod \omega(x)$.

1. **while** $k > m$ **do**
2. **if** $d_k \neq 0$ **do**
3. $d(x) = d(x) + x^{k-m}\omega(x)$
4. $k = k - 1$
5. **return**$(d(x))$

5.4.3.1 Using Special Modulus

It is easy to see that Algorithm 13 can be optimized when the modulus $\omega(x)$ is a sparse polynomial. In practice, $\omega(x)$ is used to construct the field $\mathbb{F}_{2^m}$ and must be irreducible. Irreducible polynomials with the minimum number of terms are trinomials and pentanomials. A trinomial is a polynomial with only three terms, while a pentanomial is a polynomial with only five terms. A trinomial or a pentanomial always exists for any field size $m < 1000$ [9].

The following irreducible trinomials and pentanomials are recommended in the FIPS 186-2 standard by NIST:

$$\begin{aligned}
&x^{163} + x^7 + x^6 + x^3 + 1\,,\\
&x^{233} + x^{74} + 1\,,\\
&x^{283} + x^{12} + x^7 + x^5 + 1\,,\\
&x^{409} + x^{87} + 1\,,\\
&x^{571} + x^{10} + x^5 + x^2 + 1\,.
\end{aligned}$$

The general form $\omega(x) = x^m + x^{m_1} + x^{m_2} + x^{m_3} + 1$ can be assumed for trinomials and pentanomials. Algorithm 14 performs fast modular reduction for a modulus in this special form.

Algorithm 14: Polynomial modular reduction for pentanomials

INPUT: Polynomials $d(x)$ and $\omega(x) = x^m + x^{m_1} + x^{m_2} + x^{m_3} + 1$.
OUTPUT: $d(x) \bmod \omega(x)$.

1. Set $\kappa = \lfloor m/w \rfloor$ and $\kappa_i = \lfloor (m - m_i)/w \rfloor$.
2. Set $\lambda = m \bmod w$ and $\lambda_i = (m - m_i) \bmod w$.
3. **for** $i = \lfloor \deg(d(x))/w \rfloor$ **downto** n
4. $\quad D_{i-\kappa_1} \mathrel{+}= D_i \gg \lambda_1, \qquad D_{i-\kappa_1-1} \mathrel{+}= D_i \ll (w - \lambda_1)$
5. $\quad D_{i-\kappa_2} \mathrel{+}= D_i \gg \lambda_2, \qquad D_{i-\kappa_2-1} \mathrel{+}= D_i \ll (w - \lambda_2)$
6. $\quad D_{i-\kappa_3} \mathrel{+}= D_i \gg \lambda_3, \qquad D_{i-\kappa_3-1} \mathrel{+}= D_i \ll (w - \lambda_3)$
7. $\quad D_{i-\kappa} \mathrel{+}= D_i \gg \lambda\ , \qquad D_{i-\kappa-1} \mathrel{+}= D_i \ll (w - \lambda\ )$
8. $i = n - 1$
9. $D_{i-\kappa_1} \mathrel{+}= D_i \gg \lambda_1$, **if**$(i > \kappa_1)$ **then** $D_{i-\kappa_1-1} \mathrel{+}= D_i \ll (w - \lambda_1)$
10. $D_{i-\kappa_2} \mathrel{+}= D_i \gg \lambda_2$, **if**$(i > \kappa_2)$ **then** $D_{i-\kappa_2-1} \mathrel{+}= D_i \ll (w - \lambda_2)$
11. $D_{i-\kappa_3} \mathrel{+}= D_i \gg \lambda_3$, **if**$(i > \kappa_3)$ **then** $D_{i-\kappa_3-1} \mathrel{+}= D_i \ll (w - \lambda_3)$
12. $D_{i-\kappa} \mathrel{+}= D_i \gg \lambda\ ,$ **if**$(i > \kappa\ )$ **then** $D_{i-\kappa-1} \mathrel{+}= D_i \ll (w - \lambda\ )$
13. **return**$(d(x))$

This algorithm uses the following equivalance relations for fast modular reduction.

$$\begin{aligned} x^m &\equiv x^{m_1} + x^{m_2} + x^{m_3} + 1 \bmod \omega(x)\ , \\ 1 &\equiv x^{-(m-m_1)} + x^{-(m-m_2)} + x^{-(m-m_3)} + x^{-m} \bmod \omega(x)\ , \\ 1 &\equiv x^{-\kappa_1 w - \lambda_1} + x^{-\kappa_2 w - \lambda_2} + x^{-\kappa_3 w - \lambda_3} + x^{-\kappa w - \lambda} \bmod \omega(x)\ . \end{aligned}$$

Here, the parameters $\kappa = \lfloor m/w \rfloor$, $\kappa_i = \lfloor (m - m_i)/w \rfloor$, $\lambda = m \bmod w$, and $\lambda_i = (m - m_i) \bmod w$.

5.4.3.2 Barrett Modular Reduction

The Barrett method for integers can be adapted to the polynomials over $\mathbb{F}_2$ to compute $c(x) = d(x) \bmod \omega(x)$ efficiently [7].

Quotient Estimation:

For the arbitrary integers k and k', the following equality always holds

$$\frac{d(x)}{\omega(x)} = \left(\frac{x^k}{\omega(x)}\right)\left(\frac{d(x)}{x^{k'}}\right)\left(\frac{1}{x^{k-k'}}\right).$$

This equality leads to a result similar to (5.3)

$$\frac{d(x)}{\omega(x)} = \frac{\lfloor x^k/\omega(x)\rfloor \lfloor d(x)/x^{k'}\rfloor}{x^{k-k'}} + \frac{\lfloor x^k/\omega(x)\rfloor r^{(2)}(x)}{x^k} + \frac{\lfloor d(x)/x^{k'}\rfloor r^{(1)}(x)}{\omega(x)x^{k-k'}} + \frac{r^{(1)}(x)r^{(2)}(x)}{\omega(x)x^k} \tag{5.8}$$

where $r^{(1)}(x) = x^k \bmod \omega(x)$ and $r^{(2)}(x) = d(x) \bmod 2^{k'}$.

Then, the quotient $q(x)$ of the division $d(x)/\omega(x)$ is estimated as

$$q(x) \approx \hat{q}(x) = \left\lfloor \frac{\lfloor x^k/\omega(x)\rfloor \lfloor d(x)/x^{k'}\rfloor}{x^{k-k'}} \right\rfloor . \tag{5.9}$$

Note that this estimation for polynomials is the same as the one for integers in (5.4), except the powers of two are replaced with the powers of x.

Estimation Error:

The quotient estimation in (5.9) will be exact, if the integers k and k' are chosen so that the last three terms in (5.3) are rational functions whose denominator degrees are greater than their numerator degrees. For this case, the quotients of the last three terms in (5.3) are zero and the quotient of the first term $\hat{q}(x) = \lfloor d(x)/\omega(x)\rfloor = q(x)$.

The denominators of the last three terms in (5.3) are greater than their numerators, if

$$\begin{aligned} \deg(\lfloor x^k/\omega(x)\rfloor) + \deg(r^{(2)}(x)) &< \deg(x^k), \\ \deg(\lfloor d(x)/x^{k'}\rfloor) + \deg(r^{(1)}(x)) &< \deg(\omega(x)) + \deg(x^{k-k'}), \\ \deg(r^{(1)}(x)) + \deg(r^{(2)}(x)) &< \deg(\omega(x)) + \deg(x^k). \end{aligned}$$

Let $\deg(d(x)) \geq \deg(\omega(x))$. The inequalities above always hold, if

$$k \geq \deg(d(x)), \qquad k' \leq \deg(\omega(x)).$$

Barrett Algorithm:

Algorithm 15 implements the Barrett algorithm. This algorithm is very similar to Algorithm 6. However, the powers of two are replaced with the powers of x. Also, the final correction step after Step 4 is omitted since the quotient estimation is exact.

Algorithm 15: Barrett modular reduction in $\mathbb{F}_2[x]$

INPUT: Polynomials over $\mathbb{F}_2$ $d(x)$ and $\omega(x)$.
OUTPUT: $c(x) = d(x) \bmod \omega(x)$.

1. Precompute $\hat{\omega}(x) = \lfloor x^k/\omega(x)\rfloor$ where $k \geq \deg(d(x))$.
2. $u(x) = \lfloor d(x)/x^{k'}\rfloor$ where $k' \leq \deg(\omega(x))$.

3. $\hat{q}(x) = \lfloor \hat{\omega}(x)u(x)/x^{k-k'} \rfloor$
4. $c(x) = d(x) + \hat{q}(x)\omega(x)$
5. **return** $c(x)$

Multiprecision Implementation:

The multiprecision Barrett implementation for integers in Algorithm 6 can be adapted for polynomials over $\mathbb{F}_2$ simply by replacing the powers of two with the powers of x.

Let $d(x)$ and $\omega(x)$ be polynomials such that $\deg(d(x)) < lw$ and $(n-1)w < \deg(\omega(x)) \leq nw$. Then, the integers k and k' in the Barrett method can be chosen as

$$k = lw \geq \deg(d(x)), \qquad k' = (n-1)w \leq \deg(\omega(x)).$$

to compute $d(x) \bmod \omega(x)$. Algorithm 16 gives the resulting Barrett algorithm using the notation in (5.6).

Algorithm 16: Multiprecision Barrett modular reduction in $\mathbb{F}_2[x]$

INPUT: $d(x)$ and $\omega(x)$ in $\mathbb{F}_2[x]$ where $d(x) < x^{lw}$ and $x^{(n-1)w} < \omega(x) \leq x^{nw}$.
OUTPUT: $c(x) = d(x) \bmod \omega(x)$.

1. Precompute $\hat{\omega}(x) = \sum_{i=0}^{l-n} \hat{\Omega}_i x^{iw} = \lfloor x^{lw}/\omega(x) \rfloor$.
2. $u(x) = \sum_{i=0}^{l-n} U_i x^{iw} = \lfloor d(x)/x^{(n-1)w} \rfloor$
3. $v(x) = \sum_{l-n \leq i+j \leq 2(l-n)} \hat{\Omega}_i U_j x^{i+j}$
4. $\hat{q}(x) = \sum_{i=0}^{l-n} \hat{Q}_i x^{iw} = \lfloor v/x^{l-n+1} \rfloor$
5. $c(x) = \sum_{i=0}^{n-1} D_i x^i + \sum_{i+j<n} \hat{Q}_i \Omega_j x^{i+j} \bmod x^{nw}$
6. **return**$(c(x))$

Note that only the required terms of $v(x)$ in Step 3 are computed. But this does not cause any approximation error since there is no carry propagation in the polynomial arithmetic. Step 5 is performed modulo x^{nw} since the quotient estimation is exact, and thus $c(x) = d(x) \bmod \omega(x) < x^{nw}$ in this step.

Algorithm 17 illustrates a w-bit Barrett modular reduction scheme presented in the work in [7]. In this scheme,

$$k = \deg(\omega(x)) + w - 1, \qquad k' = \deg(\omega(x)),$$

and $\lfloor d(x)/\omega(x) \rfloor < x^w$.

Algorithm 17: w-bit Barrett modular reduction in $\mathbb{F}_2[x]$

INPUT: $d(x)$ and $\omega(x)$ in $\mathbb{F}_2[x]$ such that $nw \geq \deg(\omega(x)) > (n-1)w$ and $\lfloor d(x)/\omega(x) \rfloor < x^w$.
OUTPUT: $c(x) = d(x) \bmod \omega(x)$.

1. Precompute $Q^{(1)} = \lfloor x^k/\omega(x) \rfloor$ where $k = \deg(\omega(x)) + w - 1$.
2. Find $Q^{(2)} = \lfloor (D_n x^w + D_{n-1})/x^{k' \bmod w} \rfloor$ where $k' = \deg(\omega(x))$.
3. $\hat{Q} = \lfloor Q^{(1)} Q^{(2)}/x^{w-1} \rfloor$
4. $c(x) = \sum_{i=0}^{n-1} D_i + \sum_{i=0}^{n-1} \hat{Q}\Omega_i \bmod x^{nw}$
5. **return** $c(x)$

5.4.3.3 Montgomery Modular Reduction

The analog of the Montgomery modular reduction for polynomials in $\mathbb{F}_2[x]$ is proposed in [12]. The Montgomery modular reduction for polynomials is given by $d(x)\theta^{-1}(x) \bmod \omega(x)$ where $\gcd(\omega(x), \theta(x)) = 1$ and $\omega(x)\theta(x) > d(x)$. For an efficient computation, $\theta(x)$ is chosen as a power of x preferably.

The Montgomery reduction $c(x) = d(x)\theta^{-1}(x) \bmod \omega(x)$ is given by

$$c(x) = \frac{d(x) + (d(x)\omega(x)^{-1} \bmod \theta(x))\omega(x)}{\theta(x)} \tag{5.10}$$

where $d(x) < \omega(x)\theta(x)$. This computation leads to an efficient modular reduction algorithm when $\theta(x)$ is a power of x and $\omega(x)^{-1} \bmod \theta(x)$ is precomputed.

Equation (5.10) is similar to the Montgomery computation in (5.5) given for integers, except, there is no need for an extra subtraction with modulus. This is because no carry propagation occurs in the polynomial arithmetic. Thus, $\varepsilon = 0$ in the following equation is obtained by using the Bezout's identity.

$$\begin{aligned} d(x) &= d(x)\theta(x)\hat{\theta}(x) + d(x)\omega(x)\hat{\omega}(x) \bmod \omega(x)\theta(x) \\ &= (d(x)\theta(x)\hat{\theta}(x) \bmod \omega(x)\theta(x)) + \\ &\quad (d(x)\omega(x)\hat{\omega}(x) \bmod \omega(x)\theta(x)) + \varepsilon\omega(x)\theta(x) \,. \end{aligned}$$

As a result, a derivation similar to the integer case yields Equation (5.10).

Montgomery Algorithm:

Let $d(x)$ and $\omega(x)$ be polynomials in $\mathbb{F}_2[x]$ such that $d(x) < x^{w(l-n)}\omega(x)$ and $\gcd(\omega(x), x) = 1$.

The polynomial θ can be chosen as $\theta = x^{w(l-n)}$. Then, Equation (5.10) is given by

$$c(x) = \left\lfloor \frac{d(x)}{x^{w(l-n)}} \right\rfloor + \left\lfloor \frac{(d(x)\hat{\omega}(x) \bmod x^{w(l-n)})\omega(x)}{x^{w(l-n)}} \right\rfloor$$

where $\hat{\omega}(x) = \omega(x)^{-1} \bmod x^{w(l-n)}$.

5.5 Multiplication in General Extension Fields

The extension field $\mathbb{F}_{p^m}$ elements are represented by the set of the polynomials of degree less than m with coefficients in $\mathbb{F}_p$. That is

$$\mathbb{F}_{p^m} = \{a(x) \mid a(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0,\ a_i \in \mathbb{F}_p\}.$$

Let $a(x)$ and $b(x)$ be two elements in $\mathbb{F}_{p^m}$. Let $c(x)$ be their product in $\mathbb{F}_{p^m}$. Then, $c(x)$ is defined as follows.

$$c(x) = a(x) \times b(x) \bmod \omega(x)$$

where $\omega(x)$ is a degree m irreducible polynomial over $\mathbb{F}_p$. As a result, the extension field multiplication needs two arithmetic operations:

- Polynomial multiplication over $\mathbb{F}_p$, and
- Polynomial modular reduction over $\mathbb{F}_p$.

The algorithms used in the modular multiplication of the polynomials over $\mathbb{F}_p$ will be studied in this section. However, the multiple precision representation, the addition, and the subtraction of the polynomials over $\mathbb{F}_p$ need to be discussed first.

Let a fixed w-bit word size be supported in a hardware or software implementation. Each w-bit word can store a single polynomial coefficient in $\mathbb{F}_p$, if $p < 2^w$. Let $a(x)$ be a polynomial over $\mathbb{F}_p$ and a_i be its ith coefficient. Then, each a_i is an integer stored in a single word and $a(x)$ is represented by an m-word array.

To perform the polynomial addition $c(x) = a(x) + b(x)$ or the polynomial subtraction $c(x) = a(x) - b(x)$, the corresponding coefficients of $a(x)$ and $b(x)$ are added or subtracted in $\mathbb{F}_p$ respectively. The coefficient additions and subtractions can be handled by single-word addition and subtraction operations ubiquitously found in the hardware and software implementations.

The previous section focuses on arithmetic in binary extension fields $\mathbb{F}_{2^m}$, which is a special case of the general extension field $\mathbb{F}_{p^m}$. The binary extension fields are preferred in hardware implementations due to the fact that subfield elements are easily representable using the signals logic zero and logic one. Also, the binary circuit technology makes the implementation of arithmetic operations rather straightforward. The addition and subtraction in the binary extension field can be performed simply by XOR operation and the multiplication involves shift and XOR operations.

Because the bit operations are slower in the general purpose processors, binary extension fields are not so great from the software point of view. The general purpose processors perform word level operations faster. Thus, some special classes of $\mathbb{F}_{p^m}$ called OEF are proposed to exploit this fast word level operation capability [1].

Let w denote the word size supported by the underlying system. An optimal extension field (OEF) is a finite field $\mathbb{F}_{p^m}$ where

- $p = 2^{w-1} \pm \alpha$ is a pseudo-Mersenne prime such that $\log_2 \alpha \leq \lfloor \frac{1}{2} w \rfloor$.
- An irreducible bionomial $\omega(x) = x^m - \lambda$ exists over $\mathbb{F}_p$.

In an OEF, elements are represented as degree $m-1$ polynomials as follows:

$$a(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$$

where $a_i \in \mathbb{F}_p$. Addition of the two elements $a(x)$ and $b(x)$ is given by

$$a(x) + b(x) = \sum_{i=0}^{m-1} c_i x^i,$$

where $c_i = (a_i + b_i) \bmod p$. To add two OEF elements we need at the most m coefficient subtractions where p is the subtrahend. Subtraction is done similarly. Choosing the prime p close to but smaller that the word size of the underlying hardware architecture makes it possible to use the efficient integer arithmetic instructions supported by the hardware. With such a choice, the result of a coefficient multiplication will fit into a double word, where it can be accessed and reduced efficiently.

5.5.1 Field Multiplication in OEF

The two steps of the field multiplication in OEF are as follows:

- The OEF elements $a(x)$ and $b(x)$ are multiplied.

  $$d(x) = a(x)b(x) = d_{2m-2}x^{2m-2} + \cdots + d_1 x + d_0$$

  where $d_i \in \mathbb{F}_p$. The polynomial $d(x)$ is calculated by m^2 coefficient multiplications and $(m-1)^2$ coefficient additions.
- The reduction $c(x) = d(x) \bmod \omega(x)$ is performed where $\omega(x) = x^m - \lambda$ is an irreducible binomial over $\mathbb{F}_p$. Since the binomial $\omega(x)$ has only two terms, reduction with $\omega(x)$ can be done efficiently. The terms of $d(x)$ with degree greater than $m-1$ can be given by $d_{m+i}x^{m+i}$ for $i \geq 0$. These terms can be reduced by

  $$d_{m+i}x^{m+i} = \lambda d_{m+i}x^i \bmod \omega(x)$$

  for $i = 0, 1, \cdots, m-2$.

Since the degree of $d(x)$ is at most $2m-2$, we need at most $m-1$ multiplications by λ and $m-1$ coefficient additions to obtain the reduced polynomial $c(x)$ where

$$c(x) = d_{m-1}x^{m-1} + [\lambda d_{2m-2} + d_{m-1}]\, x^{m-2} + \cdots + [\lambda d_{m+1} + d_1]\, x + [\lambda d_m + d_0] \bmod\ \omega(x).$$

The following algorithm integrates the reduction into the multiplication steps without focusing on the coefficient arithmetic operations.

Algorithm 18: OEF Modular Multiplication Algorithm

INPUT: OEF elements $a(x), b(x)$ with degree at most $m-1$. $\omega(x) = x^m - \lambda$.
OUTPUT: $c(x) = a(x)b(x) \bmod \omega(x)$.

1. **for** $i = 0$ **to** $m-1$ **do** $c_i = 0$
2. **for** $i = 0$ **to** $m-1$
3. **for** $j = 0$ **to** $m-1$
4. **if** $i+j \leq m-1$ **then** $c_{i+j} = c_{i+j} + b_i a_j$
5. **else** $c_{i+j-m} = c_{i+j-m} + b_i a_j \; w$
6. **return** $c(x)$

In Step 4 and Step 5 of Algorithm 18, we are performing coefficient multiplications and additions. If we skip the coefficient addition operation for $i+j=0$ in these steps, we end up with $(m-1)^2$ coefficient additions. The total number of coefficient multiplications is m^2+m+1 where $m-1$ of them come from the multiplication by λ. When $\omega(x)$ is selected as $\omega(x) = x^w - 2$, the coefficient multiplications by λ become simple right shift operations which can be implemented very fast. OEF's with this optimization are called Type II OEF's.

5.5.2 Coefficient Multiplication and Reductions

The coefficient multiplications and reductions can be calculated efficiently when $p = 2^{w-1} \pm \alpha$ is a pseudo-Mersenne prime not exceeding the word boundary and α is a small number. The result of the coefficient multiplication can be stored in a double word before reduction is performed. The reduction operation will reduce the result allowing it to fit into a single word. Algorithms that perform this reduction are reported in the literature. Algorithm 19 performs such a reduction operation where the α term is fixed to a negative integer.

Algorithm 19: Coefficient Reduction Algorithm

INPUT: $p = 2^{w-1} - \alpha$. Coefficient $c < p^2$.
OUTPUT: $c \bmod p$.

1. $q_0 = \lfloor c/2^{w-1} \rfloor$, $r_0 = c - q_0 2^{w-1}$
2. $r = r_0$, $i = 0$
3. **while** $q_i > 0$
4. $q_{i+1} = \lfloor q_i \alpha / 2^{w-1} \rfloor$
5. $r_{i+1} = q_i \alpha - q_{i+1} 2^{w-1}$
6. $i = i+1$, $r = r + r_i$
7. **while** $r \geq p$ **do** $r = r - p$
8. **return** r

In Step 1 of Algorithm 19, q_0 is initialized with the upper word and r_0 is initialized with the lower word of the input c. We want to reduce the upper word in one big step by taking out $q_0 2^{w-1}$. But by doing so we have taken out an extra $q_0 c$ value. We need to add this value back to the remainder. In Steps 5 and 6, we can see this effort. But, before adding this value back we further reduce it with in Steps 4 and 5. Because α is small, Step 4 is executed at the most twice. If α is selected as 1 the multiplications in Steps 3 and 4 become trivial. An OEF that supports this optimization is named as Type I.

5.6 Karatsuba–Ofman Algorithm

In this section, the fast multiplication method Karatsuba–Ofman is discussed for polynomials. This algorithms can also be used in the multiplication of large integers. In this case, x can be thought as the radix value in the multidigit representation of the integers.

Let $a_0 + a_1 x$ and $b_0 + b_1 x$ be two polynomials over a ring $\mathbb{R}$. As seen below, their multiplication using the schoolbook method

$$(a_0 + a_1 x)(b_0 + b_1 x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2$$

needs the computation of four ring products. The Karatsuba method performs this multiplication by computing only three ring products as follows

$$\begin{aligned}(a_0 + a_1 x)(b_0 + b_1 x) &= a_0 b_0 + a_1 b_1 x^2 + [a_0 b_0 + a_1 b_1 + (a_0 - a_1)(b_1 - b_0)]x \\ &= a_0 b_0 (1 + x) + a_1 b_1 (x + x^2) + (a_0 - a_1)(b_1 - b_0)x.\end{aligned}$$

This method can be generalized for arbitrary degree polynomials. Let $y = x^n$. Let $a_i(x)$ and $b_i(x)$ be polynomials with degree at the most $n - 1$. Then,

$$(a_0(x) + a_1(x)y)\,(b_0(x) + b_1(x)y) \tag{5.11}$$

is a product of the polynomials with degree at most $2n - 1$ and can be computed with the Karatsuba method using the following three half-sized products

$$a_0(x)b_0(x), \qquad a_1(x)b_1(x), \qquad (a_0(x) - a_1(x))(b_1(x) - b_0(x)). \tag{5.12}$$

Here, $a_i(x)$ and $b_i(x)$ are the coefficients of the linear polynomials in y in the ring $\mathbb{R}[x]$.

As seen, the Karatsuba method computes a product from three half-sized products. In the same fashion, it computes each of these half-sized products from three quarter-sized products. This process goes recursively. When the products get very small, the recursion stops and these small products are computed by the schoolbook method. This recursive computation constitutes a multiplication method asymptotically faster than the $\mathcal{O}(n^2)$ schoolbook method.

5.6.1 Complexity

It can be shown that the Karatsuba multiplication is $\mathcal{O}(n^{1.58})$ [10]. Let $T(n)$ denote the complexity of the multiplying polynomials with degree $n-1$. Then, the complexity of the multiplying polynomials with degree $2n-1$ is $T(2n)$. And, if the Karatsuba method is used in the computation,

$$T(2n) \leq 3T(n) + \alpha n$$

for some constant α, since the Karatsuba method uses three half-sized products plus some additions and subtractions. The recursion above implies by induction that

$$T(2^k) \leq \alpha(3^k - 2^k), \qquad k \geq 1.$$

Then, $T(n) \leq \alpha(3^{\lceil \log_2 n \rceil} - 2^{\lceil \log_2 n \rceil}) < \alpha 3^{1+\log_2 n} = 3\alpha 3^{\log_2 n} = 3\alpha n^{\log_2 3} \approx 3\alpha n^{1.58}$.

5.6.2 Number of Scalar Multiplications

Let #mul(n) denote the number of the scalar products required for the multiplication of two degree $n-1$ polynomials. As can be understood from (5.11) and (5.12), the Karatsuba method computes a product of degree $2n-1$ polynomials from the three products of degree $n-1$ polynomials. Thus,

$$\#mul(2n) = 3\ \#mul(n)$$

for the Karatsuba method. As a result, if n is a power of two,

$$\#mul(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Let n be a power of 2, the number of scalar products

$$\#mul(2n) = 2\ \#mul(n) + \#mul(n-1) - 1.$$

5.6.2.1 Integer Multiplication

To multiply two n-digit integers a and b with the Karatsuba–Ofman method, these integers are first split into the half-sized integers

$$\begin{array}{ll} a_H = (a_{n-1}, \ldots, a_{\lceil n/2 \rceil}), & a_L = (a_{\lceil n/2 \rceil - 1}, \ldots, a_0), \\ b_H = (b_{n-1}, \ldots, b_{\lceil n/2 \rceil}), & b_L = (b_{\lceil n/2 \rceil - 1}, \ldots, b_0). \end{array} \tag{5.13}$$

The integers above are made up from the higher and the lower digits of a and b. Thus, $a = a_L + a_H \beta^{\lceil n/2 \rceil}$ and $b = b_L + b_H \beta^{\lceil n/2 \rceil}$ where β is the integer base. Next, the three subproducts $f = a_L b_L$, $g = a_H b_H$, and $e = (a_L - a_H)(b_L - b_H)$. Finally, the results are combined to produce

$$d = f + g\beta^{2\lceil n/2\rceil} + (f + g - e)\beta^{\lceil n/2\rceil}. \tag{5.14}$$

Notice $f + g - e = a_L b_H + a_H b_L$ gives the sum of the cross products. Thus, the Karatsuba–Ofman method actually computes

$$d = a_L b_L + a_H b_H \beta^{2\lceil n/2\rceil} + [a_L b_H + a_H b_L]\beta^{\lceil n/2\rceil} = a \times b.$$

Algorithm 20 multiplies two integers using Karatsuba–Ofman method. In Step 1, the standard multiplication is used without any recursion, if the inputs are smaller than a threshold. Otherwise, the remaining steps are executed. First, $f + g - e = a_L b_L + a_H b_H - (a_L - a_H)(b_L - b_H)$ needs to be computed from the half-sized operands. To work with only positive operands, this term can also be computed as $f + g - e = a_L b_L + a_H b_H - s_a s_b |a_L - a_H||b_L - b_H|$ where $s_a = \text{sign}(a_L - a_H)$ and $s_b = \text{sign}(b_L - b_H)$.

Algorithm 20: Karatsuba–Ofman multiplication for integers

INPUT: n-digit integers a and b.
OUTPUT: $2n$-digit integer $d = a \times b$.

1. **if** $n \le n_{\text{threshold}}$ **then** $d = a \times b$, **return**(d)
2. Split a into $a_H = (a_{n-1}, \ldots, a_{\lceil n/2\rceil})$ and $a_L = (a_{\lceil n/2\rceil - 1}, \ldots, a_0)$.
3. Split b into $b_H = (b_{n-1}, \ldots, b_{\lceil n/2\rceil})$ and $b_L = (b_{\lceil n/2\rceil - 1}, \ldots, b_0)$.
4. $s_a = \text{sign}(a_L - a_H)$ (Use Algorithm 21.)
5. $s_b = \text{sign}(b_L - b_H)$ (Use Algorithm 21.)
6. **if** $s_a = +1$ **then** $a_M = a_L - a_H$ **else** $a_M = a_H - a_L$
7. **if** $s_b = +1$ **then** $b_M = b_L - b_H$ **else** $b_M = b_H - b_L$
8. $e = s_a s_b$ recursive-call(a_M, b_M)
9. $f =$ recursive-call(a_L, b_L)
10. $g =$ recursive-call(a_H, b_H)
11. $h = f + g - e$
12. $d = f + g\beta^{2\lceil n/2\rceil} + h\beta^{\lceil n/2\rceil}$
13. **return**(d)

The signs s_a and s_b are obtained by Algorithm 21.

Algorithm 21: Integer comparison

INPUT: k-digit integer u and l-digit integer v where $k \ge l$.
OUTPUT: $s = \text{sign}(u - v)$.

1. $s = +1$, $i = k$
2. **while** $i > l$ and $u_i = 0$ **do** $i = i - 1$
3. **if** $i = l$ **then**
4. **while** $i \ge 0$ and $u_i = v_i$ **do** $i = i - 1$
5. **if** $i \ge 0$ and $u_i < v_i$ **then** $s = -1$
6. **return**(s)

Algorithm 20 requires some multiprecision additions and subtractions. These operations are performed as shown in (5.1) and (5.2). The subtractions in Steps 6 and 7 have at the most $\lceil n/2 \rceil$-digit operands and produce a positive $\lceil n/2 \rceil$-digit result. The addition and the subtraction in Step 11 have at the most $2\lceil n/2 \rceil$-digit operands. These operations produce $h = a_L b_H + a_H b_L$. Element h is an $(n+1)$-digit positive integer since the sizes of $a_L b_H$ and $a_H b_L$ are $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$ digits. Also, the multiplications by the powers of the base β in Step 12 are nothing else than multi-digit left shifts.

5.7 Exercises

1. As shown in Section 5.2, the elements in $\mathbb{F}_p$ can be represented by the integers $\{0,1,2,\ldots,p-1\}$ and the field multiplication in $\mathbb{F}_p$ can be defined as the multiplication modulo p in $\mathbb{Z}$. Show that every non-zero field element has a multiplicative inverse according to this definition. Hint: Use the Bezout's identity for integers $u\hat{u} + v\hat{v} = \gcd(u,v)$ and investigate the case $u = p$ and $0 \leq v < p$.
2. As shown in Section 5.2, the elements in $\mathbb{F}_{p^m}$ can be represented by the polynomials over $\mathbb{F}_p$ of degree less than m. Also, the field multiplication in $\mathbb{F}_{p^m}$ can be defined as the polynomial multiplication modulo $\omega(x)$ where $\omega(x)$ is degree m irreducible polynomial over $\mathbb{F}_p$. Show that every non-zero field element has a multiplicative inverse according to this definition. Hint: Use the Bezout's identity for polynomials

$$u(x)\hat{u}(x) + v(x)\hat{v}(x) = \gcd(u(x), v(x))$$

   and investigate the case $u(x) = \omega(x)$ and $0 \leq \deg(v(x)) < m$.
3. Use the equality $\sum_{k=0}^{n-1} k\beta^{k-1}(\beta-1)^2 = n(\beta-1)\beta^n - (\beta^n - 1)$ and show that the three digit number (U,H,L) in Algorithm 2 does not overflow, if $n(\beta-1) \leq \beta^2$ where β is the integer base and n is operand size in the number of digits.
4. Use Algorithm 5 as an example and construct an efficient algorithm to reduce the integers modulo $2^{224} - 2^{96} + 1$.
5. As shown in the chapter, the Barret Algorithm for integers estimates the quotient $\lfloor d/p \rfloor$ with at most two errors, if the parameters k and k' satisfy that $k \geq \log_2 d \geq \log_2 p \geq k'$. Let these parameters be chosen such that $k \geq \log_2 d - u$ and $k' \leq \log_2 p + v$ where $k \geq k'$ still holds. Show that the quotient estimation error will be at most $2^u + 2^v$.
6. As shown in the chapter, the Barret Algorithm for polynomials over $\mathbb{F}_2$ estimates the quotient $\lfloor d(x)/\omega(x) \rfloor$ without any error, if the parameters k and k' satisfy that $k \geq \deg(d(x)) \geq \deg(\omega(x)) \geq k'$. Let these parameters be chosen such that $k \geq \deg(d(x)) - u$ and $k' \leq \deg(\omega(x)) + v$ where $k \geq k'$ still holds. What will be the error in the quotient estimation?
7. Algorithm 19 fixes the pseudo-Mersenne prime to the form of $p = 2^{w-1} - \alpha$. What changes do you need to make to this algorithm so that it will support pseudo-Mersenne primes in the form $p = 2^{w-1} \pm \alpha$.

5.8 Projects

1. Implement the recursive Karatsuba-Ofman algorithm in C for integer multiplication and polynomial multiplication in $\mathbb{F}_2$.
2. Implement the algorithms given in this chapter in an algebraic computational system (such as, Maple, Mathematica, or Matlab).

References

1. D. V. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 2000.
2. P. Barrett. Implementing the Rivest Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology—CRYPTO 86, Proceedings*, Lecture Notes in Computer Science, vol. 263, pp. 311–323. Springer, Berlin, Germany, 1986.
3. E. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, NY, 1968.
4. R. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley, Reading, MA, 1983.
5. A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In *Crypto '93*, Lecture Notes in Computer Science, vol. 773, pp. 175–186, 1994.
6. M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. *Topics in Cryptology – CT-RSA 2001*, Lecture Notes in Computer Science, vol. 2020, pp. 250–265, Springer, Berlin, Germany, 2001
7. J. F. Dhem. Efficient modular reduction algorithm in $\mathbb{F}_q[x]$ and its application to "left to right" modular multiplication in $\mathbb{F}_2[x]$. In C. D. Walter, editor, *Cryptographic Hardware and Embedded Systems – CHES 2003*, Lecture Notes in Computer Science, vol. 2779, pp. 203–213. Springer, Berlin, Germany, 2003.
8. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
9. IEEE P1363. Standard specifications for public-key cryptography.
10. D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, Third edition, 1998.
11. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
12. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.
13. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
14. R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, Boston, MA, Second edition, 1989.

15. A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.
16. A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields*. Kluwer Academic Publishers, Boston, MA, 1993.
17. V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology—CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pp. 417–426. Springer, Berlin, Germany, 1985.
18. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
19. W. W. Peterson and E. J. Weldon Jr. *Error-Correcting Codes*. MIT Press, Cambridge, MA, 1972.
20. J. Solinas. *Generalized Mersenne numbers*. Technical Report CORR 99-39, Dept. of C&O, University of Waterloo, 1999.
21. S. B. Wicker and V. K. Bhargava, editors. *Reed-Solomon Codes and Their Applications*. IEEE Press, New York, NY, 1994.

Chapter 6
Efficient Unified Arithmetic for Hardware Cryptography

Erkay Savaş and Çetin Kaya Koç

6.1 Introduction

The basic arithmetic operations (i.e., addition, multiplication, and inversion) in finite fields, $GF(q)$, where $q = p^k$ and p is a prime integer, have several applications in cryptography, such as RSA algorithm, Diffie-Hellman key exchange algorithm [1], the US federal Digital Signature Standard [2], elliptic curve cryptography [3, 4], and also recently identity-based cryptography [5, 6]. Most popular finite fields that are heavily used in cryptographic applications due to elliptic curve-based schemes are prime fields $GF(p)$ and binary extension fields $GF(2^n)$. Recently, identity-based cryptography based on pairing operations defined over elliptic curve points has stimulated a significant level of interest in the arithmetic of ternary extension fields, $GF(3^n)$.

Even though the aforementioned three popular finite fields are dissimilar mathematical structures, their elements are represented using similar data structures inside the digital circuits and computers. Furthermore, similarity of algorithms for basic arithmetic operations in these fields allows a unified module design. For example, the steps of the original Montgomery multiplication algorithm [7], which is one of the most efficient methods for multiplication in finite fields, $GF(p)$ and rings slightly differ from those of the Montgomery multiplication algorithm for binary extension fields, $GF(2^n)$ given in [8]. In addition, it is almost straightforward to extend the Montgomery multiplication algorithm for ternary extension fields, $GF(3^n)$, by essentially keeping the steps of the algorithm intact. Similarly, addition or inversion operations can be performed using similar algorithms that can be realized together in the same digital circuit.

To summarize, an arithmetic module which is versatile in the sense that it can be adjusted to operate in more than one of the three fields is feasible, provided that this extra functionality does not lead to an excessive increase in area

Sabancı University, e-mail: erkays@sabanciuniv.edu · City University of Istanbul & University of California Santa Barbara, e-mail: koc@cryptocode.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_6,

© Springer Science+Business Media, LLC 2009

and dramatic decrease in speed. Quite contrarily, a *unified* module that is capable of performing arithmetic in more than one field in the same, unified datapath brings about many advantages, one of which is the improved {area × time} product.

6.2 Fundamentals of Extension Fields

The elements of the prime finite field $GF(p)$ are the integers $\{0,1,2,...,p-1\}$ where p is an odd prime. The addition and multiplication operations in $GF(p)$ are modular operations performed in two steps:

1. Regular integer addition or multiplication, and
2. Reduction by the prime modulus p if the result of the first step is greater than or equal to the modulus.

The elements of the binary extension field $GF(2^n)$ can be represented as binary polynomials of degree less than n if polynomial basis representation is used. Analogous to the odd prime used in $GF(p)$, a binary irreducible polynomial of degree n is used to construct $GF(2^n)$. The addition in $GF(2^n)$ is simply performed by modulo-2 addition of corresponding coefficients of two polynomials. Since it is basically a polynomial addition there is no carry propagation and the degree of the resulting polynomial cannot exceed $n-1$. On the other hand, multiplication in $GF(2^n)$ is more complicated and sometimes it is beneficial to use other types of representation techniques than standard polynomial basis such as Gaussian normal basis [9]. Here, we always use polynomial basis for $GF(2^n)$ because of its suitability to the unified architecture.

Polynomial basis representation of $GF(2^n)$ is determined by an irreducible binary polynomial $p(x)$ of degree n. Given $p(x)$, all the binary polynomials of degree less than n, which has the form $A(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$, are elements of $GF(2^n)$. Multiplication in $GF(2^n)$, similar to multiplication in $GF(p)$, is performed in two steps:

1. Polynomial multiplication, followed by
2. A polynomial division of the result from Step 1 by the irreducible polynomial $p(x)$.

Similar to binary extension fields, the elements of ternary extension fields $GF(3^n)$ can be represented as (ternary) polynomials of degree at the most $n-1$, whose coefficients are from the base field $GF(3)$. In order to utilize polynomial basis for ternary arithmetic, an irreducible *ternary* polynomial $p(x)$ of degree n is needed. The addition operation in $GF(3^n)$ is polynomial addition where the corresponding coefficients of two ternary polynomials are added modulo-3 and there is no carry propagation. The multiplication is also done in two steps: a polynomial multiplication followed by reduction by the irreducible ternary polynomial $p(x)$.

6.3 Addition and Subtraction

The most fundamental arithmetic operation in finite fields and rings, on which all other arithmetic operations are based, is the addition operation. The key point to an efficient finite field arithmetic is to design fast and lightweight adder circuits. In many cryptographic applications, in order to balance the speed and area efficiency, adders utilizing redundant representation are preferred. The most basic form of redundant representation is the carry-save form in which an integer is represented as the sum of two other integers, namely $x = x_C + x_S$ where x_C and x_S are known as carry and sum components of the integer, respectively. The addition operation for carry-save representation can then be performed using full adders which have three binary inputs and two binary outputs. Full adders connected to each other in cascaded fashion can perform addition where one of the operands is in redundant form while the other is in non-redundant form.

It is possible to perform both $GF(p)$ and $GF(2^n)$ addition operation using a so-called dual-field adder (DFA) [10], which is illustrated in Figure 6.1. The DFA shown in Figure 6.1 is basically a full adder equipped with the capability of performing bit addition both with and without carry. It has an input denoted as fsel that provides this functionality. When fsel $= 1$, the dual-field adder circuit performs bitwise addition with carry which enables the circuit operating in $GF(p)$-mode. When fsel $= 0$, on the other hand, the output Cout is forced to 0 regardless of the values of the inputs. Consequently, the output S produces the result of modulo-2 addition of three binary input values. At most only two of the three binary input values of DFA can have nonzero values in $GF(2^n)$-mode.

An important aspect of designing a DFA is not to increase the critical path delay (CPD) of the circuit, which otherwise would have a negative effect in the maximum applicable clock frequency; a situation which is against the design goal of the unified modules. However, a small amount of overhead in area can be accommodated. Gate level realization of DFA shown in Figure 6.1 clearly demonstrates that there is no increase in the CPD since the two XOR gates dominate the CPD as in the case of a

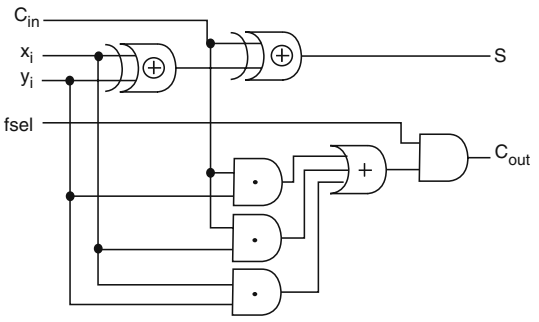

Fig. 6.1 The dual-field adder circuit.

regular full adder. Area differs slightly due to one extra input, i.e., **fsel** and additional gates that are used to suppress the carry out in $GF(2^n)$-mode. However, this increase in area is very small, and therefore tolerable, compared to two separate adders for $GF(p)$ and $GF(2^n)$ which would incur much more overhead in area if a non-unified approach were preferred.

As described above, 3×2 adder arrays in cascade are in many cases sufficient since addition operation is mostly needed in multiplications where one of the operands is always in non-redundant form as in [11]. In this case, the carry-save form is only used during the multiplication for partial product and the result of the multiplication has to be converted to non-redundant form using a carry-propagation adder after the multiplication is completed. However, when the two operands are both in carry-save redundant form, then 3×2 adder arrays in cascade cannot be used for unified addition. Instead, 4×2 adder arrays are needed to operate on both operands of redundant form. Using 4×2 adder arrays eliminates the need for conversion after multiplication, which is especially useful in elliptic curve cryptography, where there are many addition and subtraction operations in between multiplication operations.

The classical carry-save redundant representation method has one major drawback due to the difficulty of performing subtraction operation. When two's complement representation is used to facilitate the representation of negative numbers as well as subtraction operation, the carry-save representation poses certain difficulties. For example, during the subtraction of two's complement operands, a carry overflow indicates whether the result is negative or positive. Since there can be a hidden carry overflow in carry-save representation, computationally intensive operations may be needed to determine the sign of the result, which in turn incurs significant increase in CPD and area.

Avizienis [12] proposed the redundant signed digit (RSD) representation to overcome this difficulty. Arithmetic in the RSD representation is almost identical to carry-save arithmetic. An integer is still represented by two positive integers; however, this time the integer is now represented as the difference (as opposed to the sum in carry-save representation) of two other integers. An integer X, therefore, is represented by x^+ and x^-, where $X = x^+ - x^-$. As can easily be deduced from the definition of RSD, there is no need for two's complement representation to handle negative numbers and subtraction operation. The RSD is, thus, a more natural representation when both addition and subtraction operations need to be supported. This is indeed the case in elliptic curve cryptography and Montgomery multiplication and inversion algorithms. An additional benefit of RSD representation is the fact that the comparison operation in $GF(p)$-mode is now possible and efficient. Integer comparison in $GF(p)$-mode can be performed utilizing a subtraction operation. After subtracting one integer from the other, a sign test can be performed directly by checking the first nonzero bit in significant positions of the result. This is in general an easy method that can be implemented by masking the most significant bits to determine which number is greater.

Realization of RSD arithmetic is very similar to carry-save arithmetic. RSD arithmetic needs generalized full adders which are shown in Figure 6.2. As observable

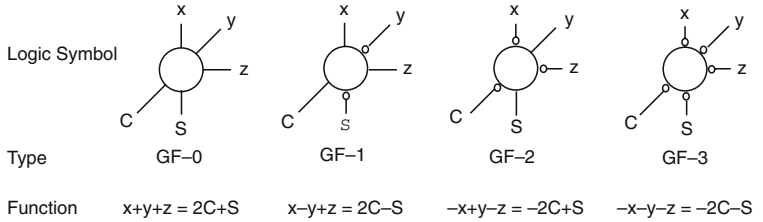

Fig. 6.2 Generalized full adders.

from Figure 6.2, GFA-0 is a conventional full adder. From the realization perspective, GFA-1, GFA-2 and GFA-3 are equivalent to GFA-0 realization in ASIC and thus there is no associated overhead in either CPD or area.

The addition of two n-bit RSD integers, X and Y, $Z = X + Y$, can be done by cascading two layers of GFAs of types 1 and 2 as shown in Figure 6.3. An additional circuitry is needed to force the digit instances of (1, 1) to (0, 0) since $1 - 1 = 0$. Subtraction of two n-bit integers, $T = X - Y$ can be realized using the same addition circuit in Figure 6.3 by swapping y^+ and y^-. The adder (or subtractor) circuit which is originally designed for $GF(p)$ arithmetic can easily be converted into a dual-field adder (or subtractor) by forcing the carry output of each GFA into 0 in $GF(2^n)$-mode.

One of the side benefits of RSD representation and associated adder structures is their suitability to a full unified arithmetic that incorporates addition/subtraction in three major finite fields, namely $GF(p)$, $GF(2^n)$ and $GF(3^n)$. Given below is the RSD representation of elements of these three fields:

1. **Prime field** $GF(p)$: Elements of prime fields can be represented as integers in binary form. Assuming that the digits are signed, the values that digits have and their corresponding representations are $\{0, 1, -1\}$ and $\{(0,0), (1,0), (0,1)\}$.
2. **Binary extension field** $GF(2^n)$: A common practice is to consider elements of binary extension field as polynomials with coefficients from $GF(2)$. This allows one to represent $GF(2^n)$ elements by simply arranging the coefficients of the

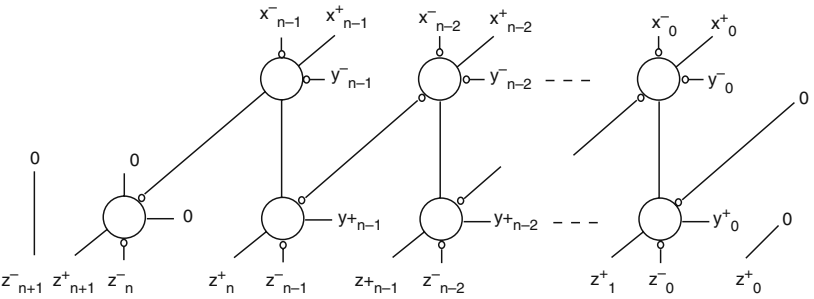

Fig. 6.3 Addition circuit with GFAs for two n-bit operands in RSD form.

polynomial into a binary string. A digit in $GF(2^n)$-mode can take the values of 1 and 0, that can be represented as $\{(0,0),(1,0)\}$.

3. **Ternary extension field** $GF(3^n)$: Elements of ternary extension fields can be considered as polynomials whose coefficients are from $GF(3)$. Thus, each coefficient can take the values $-2,-1,-,1,2$. The digit values -2 and 2 are congruent to 1 and -1 modulo 3, respectively. Therefore, the RSD representations for possible coefficient values of 0, 1, and -1 are $\{(0,0),(1,0),(0,1)\}$.

A unified adder that operates in three fields can be derived from the addition circuit in Figure 6.3. When compared to $GF(p)$-only adder, the unified adder circuit has only marginally higher CPD while the overhead in area can be higher. However, when the area cost of three non-unified adders implemented in separate datapaths far outweighs this overhead in the unified design as shown in [13].

6.4 Multiplication

In this section, we first provide the original unified Montgomery multiplication algorithm in [10], which operates only in $GF(p)$ and $GF(2^n)$. We then present a dual-radix unified multiplier in [14] where the multiplier calculates faster in $GF(2^n)$-mode than in $GF(p)$-mode. We finally discuss the support in the unified multiplier for multiplication in $GF(3^n)$.

6.4.1 Montgomery Multiplication Algorithm

In Ref. [7], Montgomery described a modular multiplication method which proved to be very efficient in both hardware and software implementations. An obvious advantage of the method is the fact that it replaces division operations with simple shift operations. The method adds multiples of the modulus rather than subtracting it from the partial result. And opposite to the subtraction of modulus in the regular modular multiplication, which can be performed after all the digits of the multiplicand are processed, the addition operation can start immediately after the least significant digit of the multiplicand is processed. Especially, the second feature accounts for the inherent concurrency in the algorithm. Refer to [7, 15, 16] for a detailed explanation of the algorithm.

Given two integers a and b, and a prime modulus p, the Montgomery multiplication algorithm computes $\bar{c} = \mathbf{MonMult}(a,b) = a \cdot b \cdot R^{-1} \pmod{p}$ where $R = 2^n$ and $a,b < p < R$ and p is an n-bit prime number. The Montgomery multiplication does not directly compute $c = a \cdot b \pmod{p}$, therefore certain transformation operations must be applied to the operands a and b before the multiplication and to the intermediate result $\bar{c}$ in order to obtain the final result c. These transformations are applied as in the following example:

$$\bar{a} = \textbf{MonMult}(a, R^2) = a \cdot R^2 \cdot R^{-1} \pmod{p} = a \cdot R \pmod{p},$$
$$\bar{b} = \textbf{MonMult}(b, R^2) = b \cdot R^2 \cdot R^{-1} \pmod{p} = b \cdot R \pmod{p},$$
$$c = \textbf{MonMult}(\bar{c}, 1) = c \cdot R \cdot R^{-1} \pmod{p} = c \pmod{p}.$$

Provided that $R^2 \pmod{p}$ is precomputed and saved, we need only a single **MonMult** operation to carry out each of these transformations. However, because of these transformation operations, performing a single modular multiplication using **MonMult** might not be advantageous even though there is an attempt to make it efficient for a few modular multiplications by eliminating the need for these transformations [17]. Its advantage, on the other hand, becomes obvious in applications requiring multiplication-intensive calculations such as modular exponentiation, elliptic curve point operations, and pairing calculations over elliptic curve points.

The Montgomery multiplication algorithm with radix-2^k for $GF(p)$ can be given as in the following:

Algorithm A

Input:	$a, b \in [1, p-1]$, p, and m
Output:	$c \in [1, p-1]$
Step 1:	$c := 0$
Step 2:	for $i = 0$ to $m-1$
Step 3:	$q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$
Step 4:	$c := (c + a_i \cdot b + q \cdot p)/2^k$

where $p'_0 = 2^k - p_0^{-1} \pmod{2^k}$. In the algorithm, the multiplier a is written with base (radix)-2^k as an array of digits a_i so that $a = \sum_{i=0}^{m-1} a_i \cdot 2^{k \cdot i}$, where m is the number of digits in a and $m = \lceil n/k \rceil$. In Step 4, the multiplicand b, the modulus p, and the partial result c enter the computations as full-precision integers. However, in the real implementations b, p, and c can be treated as multi-word integers in order to design a scalable multiplier and in each clock cycle one word of these values will be processed. One may also consider this representation as writing the multiplicand, the modulus and the partial result with digits $b^{(j)}$, $p^{(j)}$, and $c^{(j)}$ of w bits, so that $b = \sum_{j=0}^{e-1} b^{(j)} \cdot 2^{w \cdot j}$, $p = \sum_{j=0}^{e-1} p^{(j)} \cdot 2^{w \cdot j}$, and $c = \sum_{j=0}^{e-1} c^{(j)} \cdot 2^{w \cdot j}$ where $e = \lceil n/w \rceil$. Note that the base-2^w used to represent b, p, and c in Step 4 is different from the radix-2^k used to represent the multiplier a in Step 3. Note also that q, c_0, b_0, and p'_0 are all k-bit integers.

In order to avoid a possible confusion due to the usage of two different bases, we elect to refer the digits of b, p and c as words when implementing Step 4, and use the term *digit* exclusively for the multiplier a, and for b_0, p'_0, and c_0 in Step 3 when they are in the same equation with the digits of a. Digits can be easily distinguished by the subscript notation (e.g., a_i or b_0) from the superscript notation of word (e.g., $b^{(j)}$). We will also use the notation $x_{i,j}$ to denote the jth bit in the ith digit of x.

In addition, the radix of the multiplier architecture is determined by the base used to represent the multiplier a.

The Montgomery multiplication algorithm for $GF(2^n)$ is given below:

Algorithm B

Input:	$a(x), b(x)$, $p(x)$, and m
Output:	$c(x)$
Step 1:	$c(x) := 0$
Step 2:	for $i = 0$ to $m-1$
Step 3:	$q(x) := (c_0(x) + a_i(x) \cdot b_0(x)) \cdot p'_0(x) \pmod{x^k}$
Step 4:	$c(x) := (c(x) + a_i(x) \cdot c(x) + q(x) \cdot p(x))/x^k$

where $p'_0(x) = p_0^{-1}(x) \pmod{x^k}$. As one easily observes, the two algorithms are almost identical except that the addition operation in $GF(p)$ becomes a bitwise modulo-2 addition in $GF(2^n)$. Although the operands are integers in the former algorithm and binary polynomials in the latter, the representations of both are identical in digital systems. In Algorithm A, there must be an extra reduction step at the end to reduce the result into the desired range if it is greater than the modulus. On the other hand, this step is not an essential part of the algorithm and there are simple conditions that can be added to the algorithm in order to eliminate it [18, 19]; hence we intentionally exclude it from the algorithm definitions.

One can also observe that the computations performed in Step 3 are of different nature in the two algorithms and depending on the magnitude of the radix used, the part of the circuit in charge of implementing them might become very complicated. However, one can easily demonstrate that these computations can be performed in a unified circuitry for small radices.

From this point on, we will only use the notation introduced in Algorithm A for both $GF(p)$ and $GF(2^n)$ and leave polynomial notation completely out of our representation of field elements in $GF(2^n)$. Operations will be deduced from the mode ($GF(p)$ or $GF(2^n)$) in which the module is operated. The elements of both fields are represented identically in the digital systems.

6.4.1.1 Processing Unit

In this section, we explain the design details of the processing unit (PU) with radix-2, which is basically responsible for performing Steps 3 and 4 of Algorithm A:

Step 3:	$q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$
Step 4:	$c := (c + a_i \cdot b + q \cdot p)/2^k$

Since we use radix-2 for our unified multiplier for the sake of simplicity (noting that it is always possible to extend it to higher radices), the least significant bits (LSB) of the operand digits, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b+p\}$ is added to the partial result c. In Figure 6.4, the architecture of the processing unit (PU) used in the unified multiplier with $w = 2$ is illustrated. The first layer of dual-field adder deals with the addition of b to the partial result c while the second layer deals with the addition of p. The value q (binary for radix-2) calculated in Step 3 of Algorithm A determines whether the modulus p is added while the value a determines whether the multiplicand b is added to the partial result.

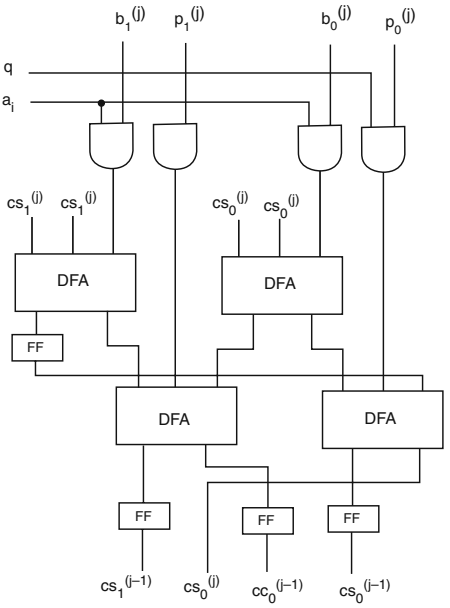

Fig. 6.4 Processing unit with radix-2 where word size $w = 2$.

As can be observed from Figure 6.4, there are flip-flops (FF) to delay some of the bit values generated during the calculations. The FF right after the first dual-field adder layer delays the most significant bit of carry from the previous word to the current word. One can think of this bit as *carry-out* from the previous word since the carry part of c is shifted one bit to the left relative to the sum part in the carry-save form. The particular arrangement of FFs at the output of the second dual-field adder layer implements right-shift operation in Step 4 of Algorithm A.

The unified architecture consists of one or (generally) more processing units (PU), identical to the one shown in Figure 6.4, organized in a pipeline. Each PU takes a digit (k-bits) from the multiplier a, the size of which depends on the radix, and operates on the words of b, c and p successively starting from the least significant words. Starting from the second cycle, it generates one word of partial result each cycle which is communicated to the next PU. After $e+1$ clock cycles, where e is the number of words in the modulus (i.e., $e = \lceil n/w \rceil$), a PU finishes its portion of work and becomes free for further computation. When the last PU in the pipeline starts generating the partial results, the control circuitry checks if the first PU is available. If the first PU is still working on an earlier computation, the results from the last PU should be stored in a buffer until the first PU becomes available again. Refer to [11] for more information about the length of the buffer to store the partial results when there is no available PU in the pipeline. In Figure 6.5 the execution graph of the Montgomery multiplication algorithm and dependencies between the processing units are illustrated.

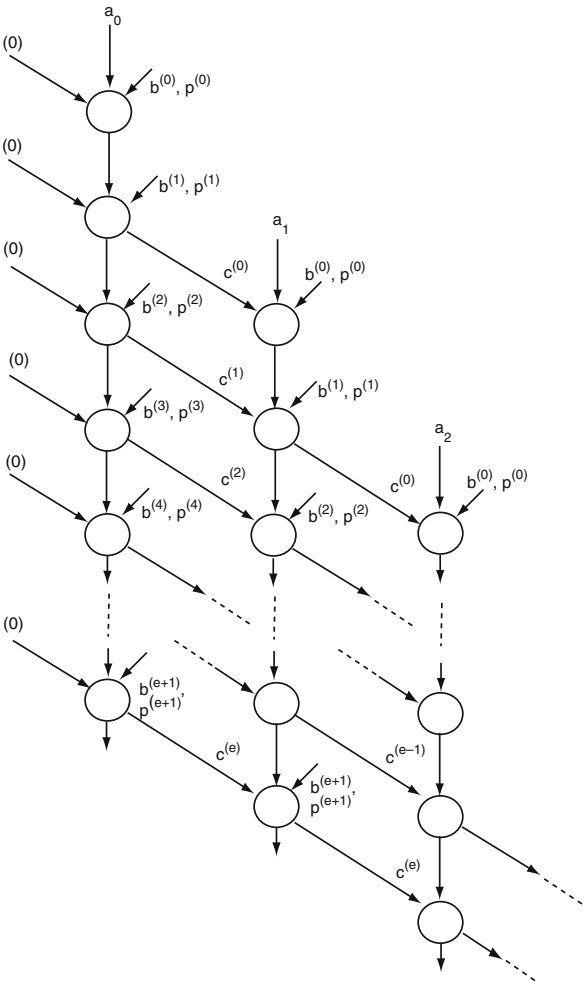

Fig. 6.5 Execution graph of Montgomery multiplication algorithm [11].

Each column in the dependency graph represents the computation which is undertaken by a PU for one digit of the multiplicand a while each circle represents the operations for one word of p, b and c. The time advances from the top to the bottom where the operation represented by a circle takes exactly one clock cycle. An example of pipeline organization with t PUs is shown in Figure 6.6.

A redundant representation (carry-save) is used for the partial result in the architecture. Thus, for the partial result we can write $c = cc + cs$, where cc and cs stand for the carry and sum part of the partial result, respectively. In addition, one must note that the length of the register for partial result, c in Figure 6.6 is twice wider than the other registers.

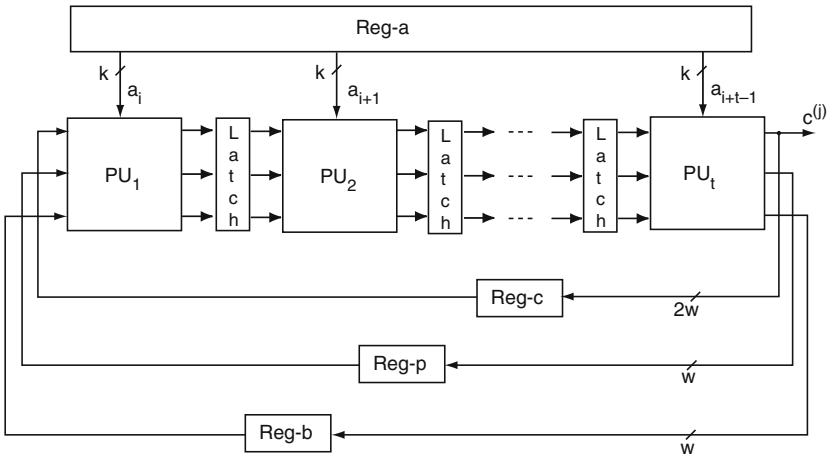

Fig. 6.6 Pipeline organization with two processing units.

Given that carry-save notation is used for the partial result and that each iteration is executed on word-by-word basis, the Algorithm A can be expressed as follows:

Algorithm A (modified)

Input: $a, b \in [1, p-1]$, p, and m
Output: $c \in [1, p-1]$, where $c = (cc, cs)$
Step 1: $cc := 0, cs := 0$
Step 2: for $i = 0$ to $m-1$
Step 3: $q := (c_0 + a_i \cdot b_0) \cdot (p'_0) \pmod{2^k}$
Step 4: for $j = 0$ to $e-1$
Step 5: $(cc^{(j)}, cs^{(j)}) := (cc^{(j)} + cs^{(j)} + a_i \cdot b^{(j)} + q \cdot p^{(j)})/2^k$

The proposed architecture allows designs with different word lengths and pipeline organizations for different values of operand precision. In addition, the area can be treated as a design constraint. Thus, one can adjust the design to the given area, and choose appropriate values for the word length and the number of pipeline stages, in accordance with it.

The propagation delay of PU is independent of word size w when w is relatively small (it increases only slightly for larger values of w due to carry-free arithmetic), and thus we assume that the clock cycle is the same for all word sizes of practical interest. The area used by registers for partial sum, operands and modulus does not change with the word or digit sizes.

The proposed scheme yields the worst performance for the case $w = m$, since some extra cycles are introduced by the PU in order to allow word-serial computation, when compared to other full-precision conventional designs. On the other hand, using many pipeline stages with small word size values brings about no advantage after a certain point. Therefore, the performance evaluation reduces to finding an optimum organization for the circuit.

ASIC standard cell realizations of both unified and non-unified ($GF(p)$-only) designs demonstrate that area overhead of the unified multiplier is only 2.75% and that there is no overhead in critical path delay [10]. Therefore, the saving in the area is significant when the unified design is compared to a hypothetical architecture that has two separate datapaths for $GF(p)$ and $GF(2^n)$ multipliers. Furthermore, this saving in area does not bring about a penalty in time performance, therefore improvement in area is identical to the improvement in metric of $\{\text{area} \times \text{time}\}$.

6.4.2 Dual-Radix Multiplier

The original unified multiplier in [10] uses radix-2 design and offers an equal performance for both $GF(p)$ and $GF(2^n)$ of the same precision in terms of clock count. For this very reason, however, the original design is not optimized since it does not take the advantage of using $GF(2^n)$, which is, in general, more efficient than $GF(p)$ in hardware implementations. Our first observation is that this situation can be remedied by putting to use the part of the circuitry which is underutilized in $GF(2^n)$ mode. This allows us to run the multiplier module in higher radix values for $GF(2^n)$ than those for $GF(p)$ at the expense of using some amount of extra gates without significantly increasing the signal propagation time.

In this section, we present the radix-(2,4) multiplier architecture introduced in [14], where the multiplier uses radix-2 in $GF(p)$-mode while it uses radix-4 in $GF(2^n)$-mode. The radix-(2,4) multiplier is in fact the first member of the dual-radix multiplier family, which also includes radix-(4, 8) and radix-(8, 16) [14]. We only include the radix-(2,4) multiplier for the sake of simplicity in explaining.

6.4.2.1 Precomputation in Montgomery Multiplication Algorithm

The dual-radix unified multiplier architecture utilizes a precomputation technique in order to decrease the critical path delay of the original unified multiplier in [10]. Note that Step 4 of the Algorithm A computes

$$c := (c_0 + a_i \cdot b + q \cdot p)/2^k$$

where division by 2^k is simply a right shift by k bits and q is calculated in the previous step. Depending on the radix value chosen for the multiplier, the k-bit digit q can be determined by the least significant digits (LSD) of b, p and c, and the current digit of a. Similarly, the multiple of b that participates in the addition is determined solely by a_i. As a result, the LSDs of the operands, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b+p, 2p, 2b, 2b+2p, \ldots\}$ is added to the partial result c. If one precomputes and stores the value of $b+p$, the calculations in Step 4 can be significantly simplified.

There are two implications of the precomputation technique. Firstly, the precomputed value must be stored, implying an increase in the register space. And secondly,

there must be a so-called selection logic to select which multiples of b and p must participate in the addition in Step 4. The selection logic can be designed in such a way that it is parallel to the PU and thus it results in no overhead in the critical path delay. On the other hand, the precomputation technique also simplifies the design since Step 4 can be performed with only one addition, once the selection logic generates its output.

6.4.2.2 Processing Unit

As pointed out earlier, a processing unit (PU) is basically responsible for performing Steps 3 and 4 of Algorithm A. Since the multiplier uses radix-2 for $GF(p)$, the LSBs of the operand digits, a_i, b_0, and c_0 will determine which one of the values in $\{0, b, p, b+p\}$ is added to the partial result c. In the case of $GF(2^n)$, multiplication is performed in radix-4. Therefore, the LSDs (least significant digits) of b, p, and c and of the current digit of a are required in order to determine q. The LSB of p is always 1, then only $p_{0,1}$, the second least significant bit of the modulus, is included in the computations. Consequently, $a_{i,1}, a_{i,0}, b_{0,1}, b_{0,0}, c_{0,1}, c_{0,0}$ and $p_{0,1}$ determine one of the following values to be added to the partial result: $\{0, b, p, b+p, x \cdot b, x \cdot p, x \cdot (b+p)\}$ (Note that $a_{i,j}$ is the jth least significant bit of ith digit of a). Multiplication by x results in shifting one bit to the left, hence it is identical to multiplication by 2. Division by x^k and 2^k are identical operations and the latter is used to denote the right-shift operation by k bits.

In Figure 6.7, the architecture of the PU used in the dual-radix multiplier is illustrated. The local control logic in Figure 6.7 contains the selection logic which

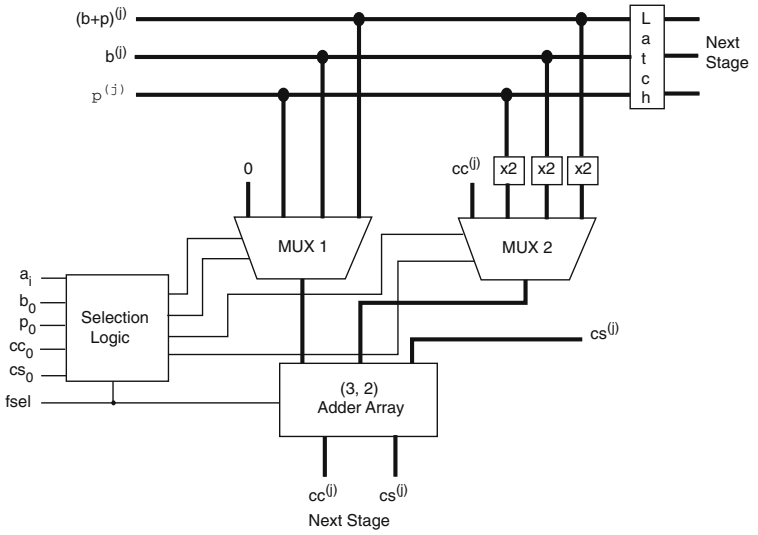

Fig. 6.7 Processing unit of dual-radix architecture with radix-2 for $GF(p)$ and radix-4 for $GF(2^n)$.

generates the signals, to determine which multiples of b and p will be in the calculations. For example, the selection signal (1011) indicates that Step 4 will be $c := (c + 3b + 2p)/2^k$. The symbols cc_0 and cs_0 in Figure 6.7 represent the least significant digits of carry and sum part of the partial result c, respectively. Note that the cary part cc of the partial result is always 0 in $GF(2)$-mode. Similarly, in $GF(p)$-mode, the multiplexer on the right-hand side always yields $cc^{(j)}$ since radix-2 is used in this mode.

6.4.3 Support for Ternary Extension Fields, GF(3^n)

The Montgomery multiplication algorithm for $GF(3^n)$, which is very similar to Algorithm B, is given below [13]:

Algorithm C

Input:	$a(x), b(x)$, $p(x)$, and m
Output:	$c(x)$
Step 1:	$c(x) := 0$
Step 2:	for $i = 0$ to $m - 1$
Step 3:	$q(x) := (c_0(x) + a_i(x) \cdot b_0(x)) \cdot p'_0(x) \pmod{x^k}$
Step 4:	$c(x) := (c(x) + a_i(x) \cdot c(x) + q(x) \cdot p(x))/x^k$

The only difference is due to the computation of $p'_0(x)$, which is $p'_0(x) = 2 \cdot p_0^{-1}(x) \pmod{x^k}$ (instead of $p'_0(x) = p_0^{-1}(x) \pmod{x^k}$ in Algorithm B).

The original unified multiplier architecture [10] utilizes two layers of (3×2) dual-field adder arrays to perform addition operations in Steps 3 and 4 of the Montgomery multiplication algorithm. This is due to the fact that multiplicand (b or $b(x)$) and modulus (p or $p(x)$) are assumed to be always in non-redundant form. This assumption can hold for elliptic curve cryptography computations, where many multiplications are needed. If the result of a multiplication which is produced in redundant form (e.g., carry-save representation), is needed for subsequent multiplications, it is immediately converted to non-redundant representation. In order to eliminate the need for conversion from redundant to non-redundant representation and the associated circuitry, all operands can be kept in redundant form throughout the entire elliptic curve computations (e.g., elliptic curve scalar point multiplication). This, however, requires using (4×2) adder arrays to perform addition(or subtraction) of two redundant form integers. Although it is laden with area and CPD overhead, one slice of (4×2) adder can easily be modified to perform one-digit addition in three fields $GF(p)$, $GF(2^n)$, and $GF(3^n)$ as explained in Section 6.3. A multiplier that can operate in three fields can be designed in the same way the original unified multiplier [10] is designed. Two important differences of the new unified multiplier from the original unified multiplier is that it has two control bits (as opposed to one in the original multiplier) to select the field mode ($GF(p)$, $GF(2^n)$, or $GF(3^n)$), and that the processing unit (PU) has now two layers of (4×2) modified-adder arrays. In addition, RSD arithmetic is employed instead of carry-save arithmetic.

In order to asses the merits of a unified multiplier that performs multiplications of three fields in the same datapath, one needs to compare the unified multiplier against a hypothetical architecture which has three separate multipliers for these three fields. The {area × CPD} metric can be used in order to figure out the balance between the saving in area and overhead in the critical path delay that the unified multiplier will have when compared to the hypothetical design. Implementations of both new unified multiplier and hypothetical design in ASIC standard cell library will demonstrate that the new unified multiplier considerably improves {area × time} metric when compared to hypothetical design [13].

6.5 Inversion

In this section, we give multiplicative inversion algorithms, which allow very fast and area-efficient unified hardware implementations. The presented algorithms are based on the Montgomery inversion algorithms given in [20]. While there are several unified inversion units reported in the literature [21–23] that compute in two fields $GF(p)$ and $GF(2^n)$ there has been no unified inversion unit proposed to operate in three fields. Therefore, we limit our discussion, which is based on the techniques and algorithms in [23], only to two basic fields, namely $GF(p)$ and $GF(2^n)$. It is, however, straightforward to extend the algorithm and its implementation to support the inversion in $GF(3^n)$.

6.5.1 Montgomery Inversion for $GF(p)$ and $GF(2^n)$

The Montgomery inversion algorithm as defined in [20] computes

$$b = a^{-1}2^n \pmod{p}, \tag{6.1}$$

given $a < p$, where p is a prime number and $n = \lceil log_2 p \rceil$. The algorithm consists of two phases: the output of Phase I is the integer r such that $r = a^{-1}2^k \pmod{p}$, where $n \leq k \leq 2n$ and Phase II is a correction step and can be modified as shown in [24] in order to calculate a slightly different inverse that can more precisely be called *Montgomery inverse*:

$$b = MonInv(a2^n) = a^{-1}2^n \pmod{p}, \tag{6.2}$$

Algorithm D
Phase I
Input: $a2^n \in [1, p-1]$ and p
Output: $r \in [1, p-1]$ and k, where $r = a^{-1}2^{k-n} \pmod{p}$ and $n \leq k \leq 2n$

1: $u := p$, $v := a2^n$, $r := 0$, and $s := 1$
2: $k := 0$

3: while $(v > 0)$
4: if u is even then $u := u/2$, $s := 2s$
5: else if v is even then $v := v/2$, $r := 2r$
6: else if $u > v$ then $u := (u-v)/2$, $r := r+s$, $s := 2s$
7: else $v := (v-u)/2$, $s := s+r$, $r := 2r$
8: $k := k+1$
9: if $r \geq p$ then $r := r-p$
10: return $r := p-r$ and k

The second phase of the Montgomery inversion algorithm simply performs $2n-k$ left (modular) shifts as a correction step to obtain $a^{-1}2^n \pmod{p}$ from $a^{-1}2^{k-n} \pmod{p}$. The left shift operations are modular in the sense that a modular reduction operation is performed whenever the shifted value exceeds the modulus.

In a similar fashion, the Montgomery inversion algorithm for $GF(2^n)$ can be given as follows:

Algorithm E
Phase I
Input: $a(x)x^n$ and $p(x)$, where $deg(a(x)x^n) < deg(p(x))$
Output: $s(x)$ and k, where $s(x) = a(x)^{-1}x^{k-n} \pmod{p(x)}$
and $deg(s(x)) < deg(p(x))$
and $deg(a(x)) + 1 \leq k \leq deg(p(x)) + deg(a(x)) + 1$

1: $u(x) := p(x)$, $v(x) := a(x)$, $r(x) := 0$, and $s(x) := 1$
2: $k := 0$
3: while $(u(x) \neq 0)$
4: if $u_0 = 0$ then $u(x) := u(x)/x$, $s(x) := xs(x)$
5: else if $v_0 = 0$ then $v(x) := v(x)/x$, $r(x) := xr(x)$
6: else if $deg(u(x)) \geq deg(v(x))$ then
$u(x) := (u(x)+v(x))/x$, $r(x) := r(x)+s(x)$, $s(x) := xs(x)$
7: else $v(x) := (v(x)+u(x))/x$, $s(x) := s(x)+r(x)$, $r(x) := xr(x)$
8: $k := k+1$
9: if $s_{n+1} = 1$ then $s(x) := s(x)+xp(x)$
10: if $s_n = 1$ then $s(x) := s(x)+p(x)$
11: return $s(x)$ and k

Additions and subtractions in the original algorithm are replaced with additions without carry in the $GF(2^n)$ version of the algorithm. Since it is possible to perform addition (and subtraction) with carry and addition without carry in a single arithmetic unit, this difference does not cause a change in the control unit of a possible unified hardware implementation. Step 6 of the proposed algorithm (where the degrees of $u(x)$ and $v(x)$ are compared) is different from that of the original algorithm. This necessitates a significant change to the control circuitry. In order to circumvent this problem we propose a slight modification in the original algorithm for $GF(p)$.

Before describing the new inversion algorithm, we first point out an important difference from the original Montgomery inversion algorithm. In Step 6 of the original Montgomery inversion algorithm two integers, u and v, are compared. Depend-

ing on the result of the comparison it is decided whether Step 6 or Step 7 is to be executed. We propose to modify Step 6 of the algorithm in a way that, instead of comparing u and v, the number of bits needed to represent them are compared. As a result of this imperfect comparisons, u may become a negative integer. The fact that u might be a negative integer may lead to problems in comparisons in subsequent iterations, therefore u must be made positive again. To do that, it is sufficient to negate r. The proposed modifications can be seen in the modified algorithm given below. Note that Algorithm F is in fact a unified algorithm and it is reduced to Algorithm E provided that all addition and subtraction operations in $GF(p)$-mode are mapped to $GF(2^n)$ additions in $GF(2)$-mode. The variable $FSEL$ is used to switch between $GF(p)$ and $GF(2)$ modes.

Algorithm F
Phase I
Input: $a2^n \in [1, p-1]$ and p
Output: $s \in [1, p-1]$ and k, where $s = a^{-1}2^{k-n} \pmod{p}$ and $n \leq k \leq 2n$

1: $u := p$, $v := a2^n$, $r := 0$, and $s := 1$
2: $k := 0$ and $FSEL := 0$ // $FSEL := 1$ in $GF(2^n)$-mode
3: if u is positive then
4: if $(bitsize(u) = 0)$ then go to Step 15
5: if u is even then $u := u/2$, $s := 2s$
6: else if v is even then $v := v/2$, $r := 2r$
7: else if $bitsize(u) \geq bitsize(v)$ then $u := (u-v)/2$, $r := r+s$, $s := 2s$
8: else $v := (v-u)/2$, $s := s+r$, $r := 2r$
9: Update $bitsize(u)$, $bitsize(v)$ and sign of u
10: else (i.e., u is negative)
11: if u is even then $u := -u/2$, $s := 2s$ $r := -r$
12: else $v := (v+u)/2$, $u := -u$, $s := s-r$, $r := -2r$
13: $k := k+1$
14: Go to Step 3
15: if $s_{n+2} = 1$ (i.e., s is negative)
16: $u := s+p$
17: $v := s+2p$
18: if $u_{n+2} = 1$ then $s := v$
19: else $s := u$
20: $u := s-p$
21: $v := s-2p$
22: if $v_{n+1} = 0$ then $s := v$
22-a: if $s_n = 1$ and $FSEL = 1$ then $s := s-p$
23: else if $u_n = 0$ then $s := u$
24: else $s := s$
25: return s and k

Changing the signs of both u and r simultaneously has the effect of multiplying both sides of the invariant $p = us + vr$ by -1. Therefore, the new invariant when

$r < 0$ is given as $\{-p = us + vr.\}$ While u and v remain to be positive integers, s and r might be positive or negative. Therefore, we need to alter the final reduction steps to bring s in the correct range, which is $[0, p)$. The range of s is $[-2p, 2p]$. As a result, we need to use two more bits to represent s and r than the bitsize of the modulus.

The value u becomes negative as a result of $u = (u - v)/2$, when $bitsize(u) = bitsize(v)$ and $v > u$ before the operation. Since $u = (u - v)/2$ decreases the bitsize of absolute value of u at least by one independent of whether the result is negative or positive, u will become certainly less than v after the negation operation. Therefore, if a negative u is encountered during the operation only Steps 11 and 12 are executed.

Note that the variable $FSEL$ is not needed for $GF(p)$-mode computations. Further, in $GF(p)$-mode $FSEL = 0$ and Step 22-a is never executed. This step becomes relevant in $GF(2)$-mode when $FSEL = 1$.

6.6 Conclusions

Unified arithmetic has gained a considerable amount of attention from the researchers and implementors working in applied cryptography. The basic premise of the unified arithmetic is that it is possible to use the same datapath for performing arithmetic operations in different fields. In this chapter, we provided the design principles of the unified arithmetic for three different fields, namely $GF(p)$, $GF(2^n)$ and $GF(3^n)$. We also pointed out the advantages of the unified arithmetic using different metrics such as area, critical path delay, operation timing, and time $\times$ area product. Although there is considerable amount of work for unified architectures for prime $GF(p)$, and binary extension $GF(2^n)$ fields, there arises a need for research on unified arithmetic units that can operate in three fields $GF(p)$, $GF(2^n)$ and $GF(3^n)$ especially with the advent of pairing-based cryptography.

6.7 Exercises

1. Obtain the truth tables for the four generalized full adders in Figure 6.2.
2. Add an additional layer of logic gates to the output of the RSD adder in Figure 6.3 to force the output (1,1) to (0,0).
3. Modify the one bit of the RSD adder circuit in Figure 6.3 so that it computes addition in three fields, namely $GF(p)$, $GF(2^n)$, and $GF(3^n)$.
4. Design (on paper) a unified multiplier of 8 bits that operates in three fields, $GF(p)$, $GF(2^n)$, and $GF(3^n)$.
5. Provide a gate level implementation (on paper) of the selection logic in Figure 6.7.
6. Design (on paper) the processing unit of a dual-radix $(4, 8)$ multiplier.
7. Obtain a Montgomery inversion algorithm by modifying the steps of Algorithm E.

6.8 Projects

1. Provide a gate level implementation of generalized full adders in Figure 6.2. Realize your implementation using ASIC standard cell library and compare areas and critical path delays of generalized full adders.
2. Implement Algorithm E in software and provide some statistics such as average number of total iterations and average number of times Steps 4, 5, 6, and 7 are executed.
3. Modify the Step 6 of Algorithm E in such a way that $u(x)$ and $v(x)$ are compared as if they are integers. Implement the algorithm in software and check if it works. Obtain the same statistics you obtained in the previous exercise. Give a comparison.

References

1. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
2. National Institute for Standards and Technology. Digital Signature Standard (DSS), *Federal Register*, 56:169, August 1991.
3. N. Koblitz, Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
4. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.
5. D. Boneh and M. Franklin. Identity-based Encryption from the Weil Pairing. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pp. 213–229. Springer-Verlag, 2001.
6. A. Shamir. Identity-Based Cryptosystems and Signature Schemes. In *Advances in Cryptology - CRYPTO 1985*, volume 196 of *Lecture Notes in Computer Science*, pp. 47–53. Springer-Verlag, 1985.
7. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
8. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). In *Proceedings of Third Annual Workshop on Selected Areas in Cryptography*, pp. 95–106, Queen's University, Kingston, Ontario, Canada, August 15–16 1996.
9. IEEE. P1363. Standard specifications for public-key cryptography. 2000.
10. E. Savaş, A. F. Tenca, and Ç. K. Koç, A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems,* Workshop on Cryptographic Hardware and Embedded Systems, pp. 277-292. Springer-Verlag, Berlin, 2000.
11. A. F. Tenca and Ç. K. Koç. "A Scalable Architecture for Montgomery Multiplication", Lecture Notes in Computer Science, 1999, 1717, pp. 94–108.
12. A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transaction Electrononic Computers*, EC(10):389–400, September 1961.

13. E. Öztürk, E. Savaş, and B. Sunar, A Versatile Montgomery Multiplier Architecture with Characteristic Three Support. Under review, 2008.
14. E. Savaç, A. F. Tenca, M. E. Ciftcibasi, and Ç. K. Koç, "Multiplier architectures for $GF(p)$ and $GF(2^k)$", *IEE Proceedings: Computers and Digital Techniques*, 151(2): 147–160, March 2004.
15. S. E. Eldridge. A faster modular multiplication algorithm. *International Journal of Computational Mathematics*, 40:63–68, 1991.
16. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
17. J.-H. Oh and S.-J. Moon. Modular multiplication method. *IEE Proceedings*, 145(4):317–318, July 1998.
18. C. D. Walter. Montgomery exponentitation needs no final subtractions. *Electronic Letters*, 35(21):1831–1832, October 1999.
19. G. Hachez and J.-J. Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Sytems*, Lecture Notes in Computer Science, No. 1965, pp. 293–301. Springer-Verlag, Berlin, 2000.
20. B. S. Kaliski Jr., The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, August 1995.
21. A. A.-A. Gutub, A. F. Tenca, E. Savaş, and Ç. K. Koç. Scalable and unified hardware to compute montgomery inverse in $GF(p)$ and $GF(2^n)$. In B. S. Kaliski Jr., Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, LNCS, pp. 485–500, Springer-Verlag Berlin, 2002.
22. E. Savaş and Ç. K. Koç, Architecture for unified field inversion with applications in elliptic curve cryptography. In Proc. vol. 3, *The 9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002*, pp. 1155–1158, Dubrovnik, Croatia, September 2002.
23. E. Savaç, M. Naseer, A. A-A. Gutub, and Ç. K. Koç. "Efficient Unified Montgomery Inversion with Multibit Shifting", *IEE Proceedings: Computers and Digital Techniques*, 152(4): 489–498, July 2005.
24. E. Savaş and Ç. K. Koç, The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, 49(7):763–766, July. 2000.

Chapter 7
Spectral Modular Arithmetic for Cryptography

Gökay Saldamlı and Çetin Kaya Koç

7.1 Introduction

Most public-key cryptosystems require resource-intensive arithmetic calculations in certain mathematical structures such as finite fields, groups, and rings. The efficient realizations of the these operations, including *modular multiplication, inversion, and exponentiation* are at the center of research activities in cryptographic engineering. Note that, being modular, these operations involve sequential reduction steps.

Spectral techniques for integer multiplication have been known for over a quarter of a century. Using the spectral integer multiplication of Schönhage and Strassen [1], large to extremely large sizes of numbers can be multiplied efficiently. Such computations are needed when computing π to millions of digits of precision, factoring, and also big prime search projects.

A naive way of utilizing the spectral techniques for modular multiplication starts with computing the multiplication using possibly Schönhage–Strassen and then a reduction in the time domain follows. This approach is preferable if the input length is large enough to meet the asymptotic crossover of Schönhage–Strassen, assuming the reduction has a constant cost. Additionally, if the naive method is used for operations involving consecutive multiplications, because of the costly forward and backward transformation computations, the asymptotic crossover of these operations would be similar to what a single modular multiplication has. Unfortunately, these crossovers are larger than the key sizes of most public-key cryptosystems; thus, in practice, the naive way is hardly used.

On the other hand, modular multiplication can be performed on the Fourier representations of integers. In such a representation, multiplications are readily available by the convolution property. Therefore, operations involving several modular multiplications can be computed efficiently.

Eczacıbaşı Embedded Design Center, e-mail: gokay.saldamli@eczacibasi.com.tr · City University of Istanbul & University of California Santa Barbara, e-mail: koc@cryptocode.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_7,

© Springer Science+Business Media, LLC 2009

In the following section, we start with introducing some preliminary notation and a formal definition of Discrete Fourier Transform (DFT) over $\mathbb{Z}_q$ (i.e., the ring of integers with multiplication and addition modulo a positive integer q). Based on this new terminology, we describe the main idea of spectral modular arithmetic including *spectral modular multiplication* (SMM) and *spectral modular exponentiation* (SME) in Section 7.3.

Section 7.4 describes methodologies for selecting the parameters for SME in order to apply the algorithm to public-key cryptography and Section 7.5 reveals how these methods can be extended to extension fields including binary and mid-size characteristic extensions. In fact, we present suitable spectrum for ECC realizations both over binary and mid-size characteristic extensions.

The chapter is closed with some final comments and discussions on the current and future research activities of the presented material.

7.2 Notation and Background

Spectral techniques are widely accepted and used in the field of digital signal processing, hence most of its existing notation and concept come from this theory. For many reasons, the signals and admissible operations on these signals of such a theory are quite different from that of a theory of computer arithmetic. For instance, when using FFT (or convolution property) for integer multiplication, first we partition the inputs into words. Note that any small perturbation in the resulting words would completely change the represented integer. On the other hand, approximations on the signal components without changing the main characteristics of the original signal are allowable in digital signal processing.

Therefore, we believe that we need a more clear notation that would permit us to have a better understanding of spectral methods and their applications to computer arithmetic-related problems. While doing this, we follow a polynomial representation instead of the standard sequence representation of digital signal processing. Such a presentation is necessary for our needs and, moreover, it states the different nature of the number-representing signals from a classical signal processing analysis.

7.2.1 Evaluation Polynomials

We start with building a new terminology that binds the polynomials over $\mathbb{Z}$ to their evaluations. A similar construction can be formulated for polynomials over the rings other than the ring of integers.

Definition 7.1. Let x and $b > 0$ be integers. If $x(t)$ is a polynomial in $\mathbb{Z}[t]$ such that $x(b) = x$, then we say $x(t)$ is an **evaluation polynomial of** x.

Remark 7.1. For ease of notation we denote the evaluation polynomials by a pair $(x,x(t))$. Sometimes we even simply use $x(t)$; the reader should be aware of polynomials in this text should always be considered with their evaluations.

Remark 7.2. Note that the positive integer b is called the **base** or **radix** in the literature. In order not to have any confusion with the frequent usage of the word "radix" for another instance in FFT theory, we prefer to use the word "base" for b.

Throughout this text we assume that b is a *fixed* positive integer and we denote the set of all evaluation polynomials over $\mathbb{Z}$ by $\mathscr{B}$. Observe that if b is fixed, there exists a natural one-to-one correspondence between $\mathscr{B}$ and $\mathbb{Z}[t]$ which is given by $(x,x(t)) \mapsto x(t)$,

In fact, the base b representation of an integer gives a special evaluation polynomial. We particularly specify these as follows:

Definition 7.2. Let

$$x(t) = x_0 + x_1 t + \ldots + x_{d-1} t^{d-1} \in \mathbb{Z}[t]$$

be an evaluation polynomial of an integer x for a fixed base b. If the coefficients of $x(t)$ satisfy $0 \leq x_i < b$ for all $i = 0,1,\ldots,d-1$, $x(t)$ is called the **base evaluation polynomial** or simply the **base polynomial** .

Example 7.1. A base $2^k, k > 0$ representation of an integer x ($(x_0 x_1 \ldots x_d)$ with $0 \leq x_i < 2^k$ for $i = 0,1,\ldots,d-1$) has the base polynomial $x(t) = x_0 + x_1 t + x_2 t^2 \ldots + x_{d-1} t^{d-1}$, where $y(t) = (x_0 + x_1 b) + 0 \cdot t + x_2 t^2 + \ldots + x_{d-1} t^{d-1}$ is one of its evaluation polynomials.

As seen in Example 7.1, the evaluation polynomial (or sequence) of an integer x is not unique. Indeed, the same integer has infinitely many different evaluation polynomials. But note that the base polynomials (i.e., base representations) are unique.

Proposition 7.1. *Let $\mathscr{B}$ denote the set of all evaluation polynomials; then $(\mathscr{B}, \oplus, \otimes)$ is a ring with the following operations;*

$$(x,x(t)) \oplus (y,y(t)) = (x+y, x(t)+y(t))$$

$$(x,x(t)) \otimes (y,y(t)) = (xy, x(t)y(t))$$

where $x(t), y(t) \in \mathbb{Z}[t]$ and $x, y \in \mathbb{Z}$.

Proof. Since base b is fixed and the structures on the components come from $\mathbb{Z}$ and $\mathbb{Z}[t]$, it is easily seen that $(\mathscr{B}, \oplus)$ is an abelian group and $(\mathscr{B}, \otimes)$ is closed. Therefore, all we need to show is that the evaluation map is well defined on the components. This is trivial because $x + y = x(b) + y(b)$, and the distribution property comes naturally from this observation. Thus, $(\mathscr{B}, \oplus, \otimes)$ is a ring with identity $(1_\oplus, 1_\otimes) = (0,1)$.

Proposition 7.2. *The map $\phi : \mathscr{B} \to \mathbb{Z}[t]$ sending $(x, x(t)) \mapsto x(t)$ is a ring isomorphism.*

Proof. Since $b > 0$ is fixed, the evaluation $x = x(b)$ is also fixed, which implies that there exists a natural subjective map from $\mathscr{B}$ to $\mathbb{Z}[t]$ sending $(x, x(t)) \mapsto x(t)$ with a zero kernel.

Definition 7.3. If $x(t)$ and $y(t)$ are evaluation polynomials for the same integer x, then we write $x(t) \sim y(t)$ and say $x(t)$ is related to $y(t)$.

Proposition 7.3. *$x(t) \sim y(t)$ is an equivalence relation.*

Proof. (i) $x(t) \sim x(t)$ since $x(b) = x(b)$
(ii) If $x(t) \sim y(t)$ then $y(t) \sim x(t)$ since $x(b) = y(b)$
(iii) If $x(t) \sim y(t)$ and $y(t) \sim z(t)$ then $x(t) \sim z(t)$ since $x(b) = y(b) = z(b)$

Proposition 7.4. *Let $\mathscr{B}$ be the set of all evaluation polynomials; then $\mathscr{B}/\sim$ is isomorphic to the ring of integers.*

Proof. Let base polynomials be the representatives of the equivalence classes of the set $\mathscr{B}$ with respect to the relation $\sim$. Since base polynomials are unique for all integers $x \in \mathbb{Z}$. The map

$$\begin{aligned} \mathbb{Z} &\to \mathscr{B}/\sim \\ x &\mapsto [(x, x(b))] \end{aligned}$$

gives the isomorphism.

Assume that $\mathbb{Z}_q$ is represented by the least residue classes $\mathscr{R} = \{0, 1, 2, \ldots, q-1\} \subset \mathbb{Z}$ (see Section 7.3.2). Evaluation polynomials defined on least residue set has a special importance for our terminology.

Definition 7.4. Let d be a positive integer. We define a **polynomial frame** as

$$\mathscr{B}_q^d = \{(y, y(t)) \in \mathscr{B} : \deg(y(t)) < d \text{ and } y_i \in \mathscr{R} \subset \mathbb{Z}\}$$

where y_i stand for the coefficients of y for $i = 0, 1, \ldots, d-1$.

Observe that

$$\mathbb{Z}_q \ncong \mathscr{B}_q^d/\sim$$

although it is correct to say $\mathscr{R}$ is equivalent to $\cong \mathscr{B}_q^d/\sim$ as a set.

On the other hand, if the frame $\mathscr{B}_q^d$ is considered, $\mathscr{B}_q^d$ is closed neither under the binary operations $\otimes$ nor $\oplus$. Thus, we remark that

$$\mathscr{B}_q^d \ncong \mathbb{Z}_q[t]/(t^d - 1).$$

However, there exists a one-to-one set map sending $(x, x(t)) \mapsto [x(t)]$ (recall that $[x(x)] = \{y(t) \in \mathbb{Z}[t] : x(t) \equiv y(t) \bmod t^d - 1\}$). Consequently, we take $\mathscr{B}_q^d$ as a simple subset of $\mathscr{B}$ without any structure on it.

7.2.2 Discrete Fourier Transform (DFT)

As computer arithmeticians, we started to build a terminology in the time domain; we represent signals by polynomials and put emphasis on their evaluations. In this section, we present the DFT as a map from the polynomial frames to a Fourier ring, and we start by introducing the Fourier rings.

Definition 7.5. Let R be a ring, the set $\mathscr{F}^d = \oplus_{i=0}^{d-1} R$ of ordered d-tuples

$$(X_0, X_1, \ldots, X_{d-1})$$

where $X_i \in S$ forms a ring with componentwise addition and multiplication (also called direct sum of rings). For notation purposes, we denote these d-tuples with polynomials (i.e., $(X_1, X_2, \ldots, X_d)$ will be written as $X_0 + X_1 t + \cdots + X_{d-1} t^{d-1}$). We named the ring $\mathscr{F}^d$ as the **Fourier ring** over R; moreover the elements are called **spectral polynomials** having **spectral coefficients** .

Remark 7.3. Throughout this text we consider only the Fourier rings over $\mathbb{Z}_q$. Therefore, we add the q subscript to our notation and denote the Fourier ring over $\mathbb{Z}_q$ by $\mathscr{F}_q^d$.

Now we can define the DFT map.

Definition 7.6. Assume that $\mathscr{B}_q^d$ is a polynomial frame and $\mathscr{F}_q^d$ is a Fourier ring over $\mathbb{Z}_q$. Let ω be a primitive d-th root of unity in $\mathbb{Z}_q$. The DFT map over $\mathbb{Z}_q$ is an invertible set map

$$\begin{aligned} DFT_d^\omega : \mathscr{B}_q^d &\to \mathscr{F}_q^d \\ (x, x(t)) &\mapsto X(t) \end{aligned}$$

defined as follows:

$$X_i = DFT_d^\omega(x(t)) := \sum_{j=0}^{d-1} x_j \omega^{ij} \bmod q \tag{7.1}$$

with the inverse

$$x_i = IDFT_d^\omega(X(t)) := d^{-1} \cdot \sum_{j=0}^{d-1} X_j \omega^{-ij} \bmod q \tag{7.2}$$

for $i = 0, 1, \ldots, d-1$. Moreover, we write

$$x(t) \quad \overset{DFT}{\longleftrightarrow} \quad X(t)$$

and say $x(t)$ and $X(t)$ are transform pairs where $x(t)$ is called a **time polynomial** and sometimes $X(t)$ is named as the **spectrum** of $x(t)$.

In the literature, DFT over a finite ring spectrum (7.1) is also known as the *Number Theoretical Transform (NTT)* . Moreover, if q has some special form such as a Mersenne or a Fermat number, the transform is named after this form i.e., *Mersenne Number Transform (MNT)* or *Fermat Number Transform (FNT)* .

Note that, unlike the DFT over the complex numbers, the existence of DFT over finite rings is not trivial. In fact, Pollard [2] mentions that the existence of primitive root d-th of unity and the inverse of d do not guarantee the existence of a DFT over a ring. He adds that a DFT exists in ring R if and only if each quotient field R/M (where M is maximal ideal) possesses a primitive root of unity. If $R = \mathbb{Z}_q$ is taken, one gets the following corollary:

Corollary 7.1. *There exists a d-point DFT over the ring $\mathbb{Z}_q$ that supports the circular convolution if and only if d divides $p-1$ for every prime p factor of q.*

Proof. We sketch the proof given in Chapter 6 of Blahut [3]. First, we cover the case where q is a prime power.

The converse is easier to prove. The DFT length d is invertible in $\mathbb{Z}_q$, if d and q are relatively prime (i.e., $dd^{-1} = 1 + kq$ for some k). Surely, any common factor of d and q must be a factor of 1, which is impossible. Moreover, any element ω having order d relatively prime to q has order that divides the Euler function $\phi(q) = (p-1)p^{m-1}$. Therefore, a d-point DFT does not exist in $\mathbb{Z}_q$ unless d divides $q-1$.

On the other hand, let p be an odd prime ($p = 2$ is trivial); then the non-units in $\mathbb{Z}_q$ form a cyclic group having order $\phi(q) = (p-1)p^{m-1}$. Let π be the generator of this group and $\omega = \pi^{bp^{m-1}}$ for any b dividing $p-1$. Since non-units in $\mathbb{Z}_q$ are cyclic, ω exists; all that remains is to show that the inverse DFT exists:

$$\begin{aligned} d^{-1} \cdot \sum_{j=0}^{d-1} X_j \omega^{-ij} \bmod q &= d^{-1} \cdot \sum_{j=0}^{d-1} \omega^{-ij} \sum_{j'=0}^{d-1} x'_j \omega^{-ij'} \bmod q \\ &= d^{-1} \cdot \sum_{j'=0}^{d-1} x'_j \sum_{j=0}^{d-1} \omega^{-i(j'-j)}. \end{aligned}$$

The sum on i is equal to d if $j' = j$, while if j' is not equal to j, then the geometric series summation becomes $(1-\omega^{-(j'-j)d})/(1-\omega^{-(j'-j)})$, which is zero since $j' - j \not\equiv 0 \pmod{q}$. Therefore,

$$d^{-1} \cdot \sum_{j=0}^{d-1} \omega^{-ij} \bmod q = d^{-1} \cdot \sum_{j=0}^{d-1} x_i (d\delta_{jj'}) \bmod q = x_i$$

as desired.

Now, let $q = p_1^{m_1} p_2^{m_2} \dots p_r^{m_r}$. The use of the Chinese remainder theorem guarantees the existence of a d-point DFT in $\mathbb{Z}_q$ if and only if d-point DFT exists in each factor ring, which is equivalent to, say, d divides $p_i - 1$ for all $i = 1, 2, \dots, r$.

Example 7.2. In general, longer length DFTs are of utmost importance in many applications. Obviously, Corollary 7.1 states that DFTs over integer rings mostly suffer

from short lengths. For instance, in the fairly large ring $\mathbb{Z}_{2^{31}+1}$, one can only define a 2-point transform since $2^{31}+1 = 3 \cdot 715827883$, though, in Section 7.4.2 we describe some solutions to overcome such problems.

7.2.3 Properties of DFT: Time–frequency dictionary

In the previous section, we relate the time and spectral polynomials by the DFT map; it is also possible to relate operations (formally we mean functions) in a similar manner. In other words, we relate a pair of maps ϕ and Φ defined on time and spectral polynomials respectively, if DFT map commutes with them. The next definition prepares a formal setup for this discussion.

Definition 7.7. Let ϕ and Φ be operations on time and spectral domains, respectively. We write

$$\phi \quad \xleftrightarrow{DFT} \quad \Phi$$

and say ϕ and Φ are transform pairs on $x(t)$ and sometimes declare that **the map** DFT_d^ω **respects the operation** ϕ on a point $x(t)$ if the following diagram commutes

$$\begin{array}{ccc} \mathscr{B}_q^d & \xrightarrow{DFT} & \mathscr{F}_q^d \\ \downarrow{\scriptstyle \phi} & & \downarrow{\scriptstyle \Phi} \\ \mathscr{B} & \xleftarrow{IDFT} & \mathscr{F}_q^d \end{array}$$

Equivalently, if the following equation is satisfied

$$\phi(x(t)) = IDFT_d^\omega \circ \Phi \circ DFT_d^\omega(x(t)). \tag{7.3}$$

Theorem 7.1. *(Fundamental) Let ϕ and Φ be operations on time and spectral domains respectively. The condition*

$$\phi(x(t)) \in \mathscr{B}_q^d$$

is necessary and sufficient for functions ϕ and Φ to be transform pairs on a point $x(t) \in \mathscr{B}_q^d$. We say an **overflow occurs** *for those cases in which $\phi(x(t)) \notin \mathscr{B}_q^d$.*

Proof. Let there exists a DFT map $DFT_d^\omega : \mathscr{B}_q^d \to \mathscr{F}_q^d$. By definition, $IDFT_d^\omega \circ \Phi \circ DFT_d^\omega(x(t))$ is an element of $\mathscr{B}_q^d$, hence $\phi(x(t))$ must be an element of $\mathscr{B}_q^d$.

In general, not having a nice domain, DFT does not globally commute with such function pairs. However, DFT respects various operations locally. Linearity, convolution and time–frequency shifting are some of these operations.

In the literature, such operations which are referred as the properties of DFT are essential for a better understanding of the nature of the transform. In fact, because of

these properties, the Fourier transform becomes a powerful tool for applied sciences. We refer the reader to [4] for a general review of these properties.

In a finite ring setting, the existence conditions of these properties are quite different from a theory over complex numbers. In here, we present various properties and further state the existence conditions of the two most important, namely, linearity and convolution. We start with some notations:

Notation 1 *Let ω be a principal d-th root of unity, and $\Gamma(t)$ and $\Omega(t)$ be the spectral polynomials with coefficients consisting of negative and positive powers of ω respectively, as follows*

$$\Omega(t) = 1 + \omega^1 t + \omega^2 t^2 + \ldots + \omega^{(d-1)} t^{(d-1)} ,$$
$$\Gamma(t) = 1 + \omega^{-1} t + \omega^{-2} t^2 + \ldots + \omega^{-(d-1)} t^{(d-1)} .$$

Notation 2 *Let $a \in \mathbb{Z}$ be a constant number, a degree d polynomial with all of its coefficients equal to a (i.e., $a(t) = a + at + at^2 + \ldots + at^d$) is denoted by $a(t)$.*

Time and frequency shifts: Time and frequency shifts correspond to circular shifts when working with finite-length signals. Let $x(t) = x_0 + x_1 t + \ldots + x_{d-1} t^{d-1}$ and $X(t) = X_0 + X_1 t + \ldots + X_{d-1} t^{d-1}$ be a transform pair. The *one-term right circular shift* is defined as

$$\begin{array}{c} x(t) \circlearrowright 1 := x_{d-1} + x_0 t + \ldots + x_{d-2} t^{d-1} \\ \Updownarrow DFT \\ X(t) \odot \Omega(t) \end{array}$$

where $\odot$ stands for componentwise multiplication. Similarly, one performs the *one-term left circular shift* by multiplying the coefficients of $X(t)$ with negative power sequence of the principal d-th root of unity:

$$\begin{array}{c} x(t) \circlearrowleft 1 := x_1 + x_2 t + \ldots + x_{d-1} t^{d-2} + x_0 t^{d-1} \\ \Updownarrow DFT \\ X(t) \odot \Gamma(t) \end{array}$$

An arbitrary circular shift can be obtained by applying consecutive one-term shifts or using a proper ω power sequence. For instance, s-term circular left shift ($0 \leq s \leq d-1$) is achieved by multiplying $X(t)$ with $\Gamma_s(t) = 1 + \omega^{-s} t + \omega^{-st} t^2 + \ldots + \omega^{-s(d-1)} t^{d-1}$, componentwise.

Sum of sequence and first value: The sum of the coefficients of a time polynomial equals the zeroth coefficient of its spectral polynomial. Conversely, the sum of the spectrum coefficients equals d^{-1} times the zeroth coefficient of the time polynomial (i.e., $x_0 = d^{-1} \cdot \sum_{i=0}^{d-1} X_i \omega^{-i}$ and $X_0 = \sum_{i=0}^{d-1} x_i \omega^i$ as seen in Figure 7.1).

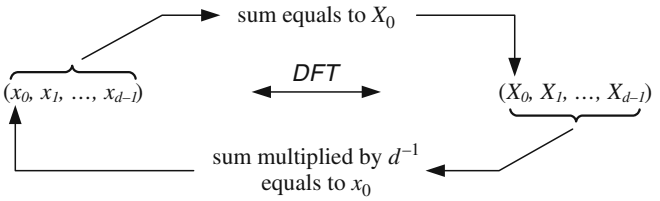

Fig. 7.1 Sum of coefficients and first coefficient.

Left and right logical shifts. Using the above properties, it is possible to achieve the logical left and right digit shifts. We begin with the one-term left shift operation. Let $x_0(t)$ be equal to $x_0 + x_0 t + \ldots + x_0 t^{d-1}$ (see Notation 2) then

$$\begin{array}{c} x(t) \ll 1 = (x(t) - x_0)/t = x_1 + x_2 t + \ldots + x_{d-1} t^{d-2} \\ \Big\updownarrow DFT \\ (X(t) - x_0(t)) \odot \Gamma(t) \end{array}$$

The right shifts are similar, where one then uses the $\Omega(t)$ polynomial instead of $\Gamma(t)$.

Using the fundamental Theorem 7.1, it is easily seen that time–frequency shifts and right/left shifts are globally respected by the DFT map. On the other hand, linearity and convolution properties are respected locally. We start by giving an example of overflow and then turn our attention back to state the conditions when these two properties are satisfied.

Example 7.3. Let ϕ_y be an operation such that

$$\phi_y : x(t) \mapsto x(t) + y(t) \text{ for all } x(t) \in \mathscr{B}_5^4$$

where $y(t) = 3 + t + t^2 + 4t^3$ is a base $b = 2$ evaluation polynomial for $y = 19$. Assume that the DFT map is a 4-point map over $\mathbb{Z}_5$, i.e., $DFT_4^\omega : \mathscr{B}_q^d \to \mathscr{F}_q^d$. Notice that the addition operation over the Fourier ring,

$$\Phi_y : X(t) \mapsto X(t) + Y(t) \text{ for all } X(t) \in \mathscr{F}_5^4$$

is a transform pair of ϕ_y on points $x(t)$ where

$$\phi_y(x(t)) = x(t) + y(t) \in \mathscr{B}_5^4 \subset \mathscr{B}. \tag{7.4}$$

Obviously, not all $x(t) \in \mathscr{B}_5^4$ satisfies Equation (7.4); for instance, if $x(t) = 3 + t^2 + t^3$ is an evaluation polynomial for $x = 15$, $\phi_y(x(t)) = x(t) + y(t) = 6 + t + 2t^2 + 5t^3$ gives an evaluation polynomial for 56 but

$$DFT_d^\omega \circ \Phi \circ DFT_d^\omega(x(t)) = 1 + t + 2t^2 \neq \phi_y(x(t)).$$

Therefore, we say ϕ_y and Φ_y are not transform pairs on $x(t) = 3 + t^2 + t^3$.

Observe that one gets the linearity property if the DFT map respects all the elements of the set $\{\phi_y : \text{for all } y(t) \in \mathscr{B}_q^d\}$ and its λ scaling for some $\lambda \in \mathbb{Z}_q$. Although the DFT map is a global group homomorphism over the additive group of complex numbers, it does not respect addition over finite-ring spectrum. The next proposition states that on a convex or a more regular subset of $\mathscr{B}_q^d$ the DFT map respects the single addition operation.

Proposition 7.5. *Let D be a subset of $\mathscr{B}_q^d$ such that $D = \{x(t) : x_i < q/2\}$ and ϕ_y be an addition operation on D for any $y \in D$. The DFT map respects ϕ_y on D.*

Proof. Let $\phi_y : D \mapsto \mathscr{F}_q^d$ be the addition map for any $y \in D$. Since $\phi_y(x(t)) = x(t) + y(t) \in \mathscr{B}_q^d$ for all $x(t)$ in D, using Theorem 7.1, DFT respects ϕ_y.

Next, we formally state when a DFT map respects the convolution operation. We start with a lemma:

Lemma 7.1. *Suppose that $(x(t),x)$ and $(y(t),y)$ are base b polynomials in the frame $\mathscr{B}_b^s$ with $s = \lceil d/2 \rceil$. The product $(z(t),z) = (x(t),x) \otimes (y(t),y)$ belongs to $\mathscr{B}_q^d$ where $q > sb^2$.*

Proof. Let $x(t) = x_0 + x_1 t + \ldots + x_{d-1} t^{d-1}$ and $y(t) = y_0 + y_1 t + \ldots + y_{d-1} t^{d-1}$ be polynomials such that $\deg(x(t)) + \deg(y(t)) < d$ and $0 \leq x_i, y_i < b$ for some $b > 0$. Without loss of generality, assume $\deg(x(t)) \geq \deg(y(t))$. If $z(t) = x(t)y(t)$, then the coefficients of $z(t)$ can be written as follows:

$$z_k = \sum_{k=i+j} x_i y_j, \qquad k = 0, 1, 2, \ldots, d-1.$$

Notice that z_k can be found by adding at the most $\deg(y(t)) + 1$ nonzero terms, but since $\deg(y(t)) + 1 \leq \lceil d/2 \rceil$, letting $s = \lceil d/2 \rceil$ gives

$$z_k \leq (\deg(y(t)) + 1) \cdot b \cdot b \leq s \cdot b^2$$

Thus, choosing $q > s \cdot b^2$ gives the result.

The following result gives the condition when the d-point DFT map respects the convolution of two elements of a frame $\mathscr{B}_q^d$.

Theorem 7.2. *Let $DFT_d^{\omega} : \mathscr{B}_q^d \rightarrow \mathscr{F}_q^d$ be a d-point DFT map and $D = \mathscr{B}_b^s$ be a subset of $\mathscr{B}_q^d$ such that $s = \lceil d/2 \rceil$ and $b^2 s < q$ for an integer $b > 0$. The DFT map respects the convolution map ϕ_y on $y(t)$ for any $y(t)$ in D where*

$$\phi_y : x(t) \mapsto x(t) \cdot y(t) \text{ for all } x(t) \in D$$

Proof. Let $\phi_y : D \mapsto \mathscr{F}_q^d$ be the multiplication map for some $y(t) \in D$. By Lemma 7.1, the product $\phi_y(x(t)) = x(t) \cdot y(t) \in \mathscr{B}_q^d$ for all $x(t)$ in D. Therefore, using Theorem 7.1, DFT respects ϕ_y.

7.3 Spectral Modular Arithmetic

7.3.1 Time Simulations and Spectral Algorithms

In the previous section, we stated the conditions when the convolution and addition properties are respected by the DFT map over a finite ring spectrum. Since an algorithm is a combination of some primitive operations, starting with an example, we bring up the notion of algorithm pairs that are respected by DFT maps.

Example 7.4. Consider an algorithm in time domain performing the following operations:

Input: $x(t), y(t) \in \mathbb{Z}[t]$ polynomials of degree d
Output: $z(t) := x(t)(5y(t) + x(t)) - 3x(t)$

1: $z(t) := x(t) + 5y(t)$
2: $z(t) := x(t) \cdot z(t)$
3: $z(t) := z(t) - 3x(t)$
4: **return** $z(t)$

Whenever the DFT map respects the above algorithm, a dual algorithm operating in the spectrum can be furnished as follows:

Input: $X(t), Y(t) \in \mathbb{Z}[t]$ polynomials of degree d
Output: $Z(t) := X(t) \odot (5Y(t) + X(t)) - 3X(t)$

1: $Z(t) := X(t) + 5Y(t)$
2: $Z(t) := X(t) \odot Z(t)$
3: $Z(t) := Z(t) - 3X(t)$
4: **return** $Z(t)$

Once again we can relate these two objects using the DFT map and write

$$\text{Algorithm 1} \quad \xleftarrow{\quad DFT \quad} \quad \text{Algorithm 2}$$

Observe that when the inputs of Algorithm 7.4 and 7.4 agree, a parallel run produces the agreeing intermediate and final results. We name Algorithm 7.4 as the **time simulation** of the **spectral algorithm** (i.e., Algorithm 7.4).

Note that our primary interest in spectral techniques is to make use of the convolution property for calculating modular multiplications. An algorithm involving several multiplications benefits most from such a motivation. For instance, the encryption algorithm RSA [5] over some integer ring has such a nature but since these

multiplications are modular, one has to deal with reductions. In the following sections, first, we describe a time simulation for modular reduction. Secondly, we translate the time simulation into a finite ring spectrum using the properties of DFT and finally, we analyze the minimal domains (i.e., smallest rings) in which our spectral algorithms work.

7.3.2 Modular Reduction

Before introducing the notion of spectral reduction, we need to make a few points clear about the modular arithmetic over the ring of integers;

In calculations of integers involving division it often happens that we are interested in the remainder, but not in the quotient. Those numbers having the same remainder when divided by a fixed number n are called congruent, to be more formal:

Definition 7.8. Let $n > 0$ be a fixed integer. We say x **is congruent to** y **modulo** n and write

$$y \equiv x \bmod n \quad \text{if } n \text{ divides } (y-x). \tag{7.5}$$

From the division algorithm we know that for each $x \in \mathbb{Z}$ there is an equation

$$x = nq + r, \quad \text{for some } q \in \mathbb{Z} \text{ and } 0 \leq r \leq n$$

This means that each $x \in \mathbb{Z}$ can be assigned to one of the elements of the set $\{0,1,2,\ldots,n-1\}$. This set is called the **least residues** $\bmod\ n$ and it is clear that no two of the elements are congruent to each other $\bmod\ n$. We define the modular reduction as follows:

Definition 7.9. Let $n > 0$ be a fixed integer. We say y is the modular reduction of x modulo n and write

$$y = x \bmod n$$

where y is a least residue $\bmod\ n$.

Remark 7.4. The expressions "$y = x \bmod n$" and "$y \equiv x \bmod n$" have different meanings. Observe that the first one with "=" states that y is in the range $[0, n-1]$.

The equivalence on $\mathbb{Z}$ defined by the relation (7.5) partitions $\mathbb{Z}$ into n blocks, called the residue classes of $\mathbb{Z}$ modulo n. In fact, $\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z}$ is the set of these residue classes. If we denote the residue class modulo n containing y by $\bar{y}$, then the $\mathbb{Z}_n$ can be seen as the ring having the following n elements $\bar{0}, \bar{1}, \ldots, \overline{(n-1)}$. For instance, when $n = 2$, the residue classes are the set of even and odd numbers.

While performing computations such as modular exponentiation, in order to have some computational advantage, sometimes exact modular reduction calculations can be postponed for the intermediate values [6]. As long as these values belong to the correct residue classes, such modifications do not tend to misleading modular

reductions. Now, we stretch the definition of the modular reduction for ease of our construction.

Definition 7.10. Let $\varepsilon > 0$ be an integer We call the set

$$F(\varepsilon) = \{y \in \mathbb{Z} : 0 \leq y < \varepsilon\}$$

the **integer frame of radius** ε.

Definition 7.11. Let $x, n > 0$ and $\varepsilon \geq n$ be integers. Then the elements of the set

$$\{y : y \in F(\varepsilon) \text{ and } y \equiv x \bmod n\}$$

are called as the **almost modular reductions of** x **with respect to the modulus** n.

Example 7.5. Let $\varepsilon = 12$ then $F(\varepsilon) = [0, 12)$ and the set of almost modular reductions of $x = 1$ with respect to modulus $n = 3$ is $\{1, 4, 7, 10\}$.

The choice of the radius ε completely depends on the nature or needs of the problem. Most of the time, the reductions are followed by a squaring or a multiplication. Therefore, as ε gets larger the operand sizes of the succeeding operations increase. Obviously $\varepsilon = n$ is the optimal choice in this sense. But, as we pointed out earlier, we are after some approximations of the optimal solution for some obvious reasons. In other words, we are looking for some small ε such that, after finding an element of almost modular reduction set, deducing the exact modular reduction has to be simple. Indeed, that is why it is appropriate to use the adjective "almost" to describe the elements of this set.

7.3.3 Spectral Modular Reduction

In this section, we give a formal definition for the spectral modular reduction and build up the necessary terminology for a better understanding of the algorithms in the spectrum. We return to our main objects: the set of evaluation polynomials, $\mathscr{B}$, and its subsets.

Proposition 7.6. *The evaluations of the polynomials in* $\mathscr{B}_r^d$ *form an integer frame* $F(\varepsilon)$ *in* $\mathbb{Z}$ *where* $\varepsilon = (r-1) + (r-1)b + (r-1)b^2 + \ldots + (r-1)b^{d-1}$.

Proof. It is easily seen that the polynomial $x(t) = (r-1) + (r-1)t + (r-1)t^2 + \ldots + (r-1)t^{d-1} \in \mathscr{B}_r^d$ attains the maximum evaluation value at base b which is the integer $(r-1) + (r-1)b + (r-1)b^2 + \ldots + (r-1)b^{d-1}$. The evaluation of the zero polynomial obviously gives the minimum value.

Definition 7.12. Let $n(t)$ be a base b polynomial of n with degree $d-1$ and $\mathscr{B}_r^d$ be a polynomial frame for some $r \geq b$. The elements of the set

$$\mathscr{A} = \{(y, y(t)) : y \equiv x \bmod n \text{ and } y(t) \in \mathscr{B}_r^d\}$$

are called **almost spectral reductions of the evaluation polynomial** $(x, x(t))$ **with respect to** $(n, n(t))$.

Lemma 7.2. *Let* $\mathscr{A}$ *be the set of all the almost spectral modular reductions of* $(x, x(t))$ *with respect to* $(n, n(t))$. *If* $y(t)$ *is the base polynomial for* $y = x \bmod n$, *then* $(y, y(t)) \in \mathscr{A}$.

Proof. If $y(t) = y_0 + y_1 t + \ldots + y_{d-1} t^{d-1}$ is the base polynomial for $y = x \bmod n$, then $0 \leq y_i < b$ for all $i = 0, 1, \ldots, d-1$. Since $r \geq b$, $y(t) \in \mathscr{B}_b^d \subset \mathscr{B}_r^d \Rightarrow (y, y(t)) \in \mathscr{A}$.

Definition 7.13. We call the base polynomial $y(t)$ of $y = x \bmod n$ as the **spectral (modular) reduction** of $(x, x(t))$ with respect to $(n, n(t))$ and we simply write

$$y(t) = x(t) \bmod n(t).$$

Moreover the expression

$$y(t) \equiv x(t) \bmod n(t)$$

mean n divides the evaluation of $(x(t) - y(t))$ at base b.

The spectral reduction defined in the time domain can be viewed as a projection of the usual modular operation in $\mathbb{Z}$ to the set of (evaluation) polynomials. Clearly, it is defined over the polynomials but it is different from the standard modular reduction in $\mathbb{Z}[t]$. To indicate this difference, in place of "mod" we choose to use "mods".

Similar objects can be defined for spectral polynomials; however, we note that unlike time polynomials, evaluation of spectral polynomials do not have any special meaning that serves our needs. To be specific, for a spectral polynomial $X(t)$, $X(b)$ does not have a special meaning, where $x(b)$ mostly represents a meaningful integer data. Therefore, our derivation for spectral polynomials is a notational continuation of the notation that is developed for time polynomials.

Definition 7.14. Let $x(t)$ be a base polynomial for $b > 0$ of an integer x. We call the spectral polynomial $X(t)$, the transform pair of $x(t)$, the **spectral base polynomial**.

Definition 7.15. Let $y(t)$ be an almost spectral reduction of $x(t)$ with respect to $n(t)$ in some frame $\mathscr{B}_r^d$. The spectral polynomial $Y(t)$, transform pair of $y(t)$, is called the **almost spectral reduction of** $X(t)$ **with respect to** $N(t)$, where $(X(t), x(t))$ and $(N(t), n(t))$ are transform pairs.

Definition 7.16. We call the base polynomial $Y(t)$ in the spectrum the **spectral modular reduction of** $X(t)$ **with respect to** $N(t)$ and we write

$$Y(t) = X(t) \bmod N(t).$$

Moreover the expression

$$Y(t) \equiv X(t) \bmod N(t)$$

means n divides the evaluation of IDFT$((X(t) - Y(t))$ at base b.

7.3.4 Time Simulation of Spectral Modular Reduction

The spectral reduction can easily be achieved by deducing the base polynomial of the usual modular reduction (i.e., $y = x \bmod n$) but with such an approach one needs to perform classical modular reduction routines, which do not have simple spectral meanings. Our next step is to give a description of an algorithm that computes an almost spectral reduction of an evaluation polynomial.

Instead of a direct reduction method, here we present an algorithm of Montgomery type [7], which allows efficient implementation of modular arithmetic operations without explicitly carrying out the classical modular reductions. In fact, it replaces the modular reduction by a multiplication and some trivial shifts. The trick is, instead of attacking to compute "$x \bmod n$" directly, it proposes to derive it after performing a related computation

$$x \cdot \tau^{-1} \bmod n$$

for $\tau > n$ and $\gcd(n, \tau) = 1$. At first glance, this seems computationally pointless because of the inversion involved but the selection of τ changes this first impression drastically. After giving some related notation, with Algorithm 7.3.4, we employ such a methodology.

Notation 3 *The polynomial product $x(t) \cdot t^e$ is denoted by $x^e(t)$, so in this context,*

$$x^{-e}(t) := x(t) \cdot t^{-e}$$

Time Simulation of Spectral Reduction Algorithm
Suppose that n and b are positive numbers with $\gcd(b, n) = 1$, $(n, n(t))$ is the base evaluation polynomial of degree $d-1$ and $(x, x(t)) \in \mathscr{B}_u^e$ where $e \geq d$ and $u \geq 0$.
Input: $x(t)$ and $n(t)$.
Output: $y(t)$, an almost spectral reduction of $x^{-e}(t)$ with respect to $n(t)$.

1: Compute $\underline{n} = \delta n$ where $\underline{n}_0 = 1$ and $|\underline{n}_i| < b/2$
2: $y(t) := x(t)$
3: $\alpha := 0$
4: **for** $i = 0$ **to** $e - 1$
5: $\quad \beta := -(y_0 + \alpha)$ **rem** b
6: $\quad \alpha := (y_0 + \alpha + \beta)$ **div** b
7: $\quad y(t) := y(t) + \beta \cdot \underline{n}(t) \bmod q$
8: $\quad y(t) := (y(t) - y_0)/t$
9: **end for**
10: $y(t) := y(t) + \alpha(t)$, for base polynomial $(\alpha, \alpha(t))$
11: **return** $y(t)$

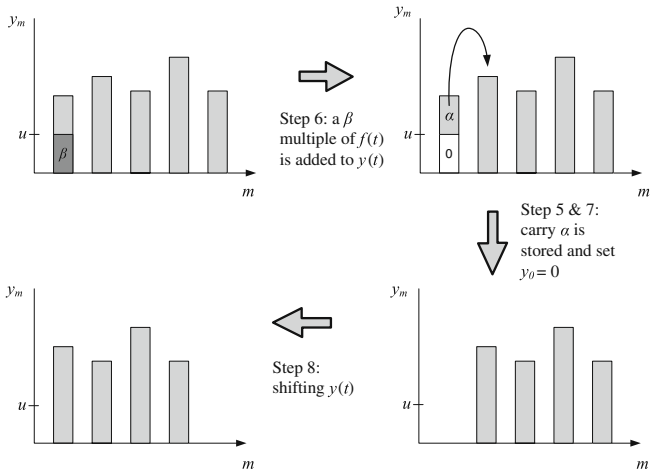

Fig. 7.2 Illustration of the reduction of a single coefficient.

In Figure 7.2, we demonstrate the reduction of a single coefficient. Once the parameters α and β are calculated from the least significant coefficient, a β multiple of the modulus $\underline{n}(t)$ is added to the sum; carry passed to the next coefficient and finally the sum is shifted.

Algorithm 7.3.4 reduces the degree while reducing the radius (i.e., the magnitude of the coefficients). In fact, one can clearly compute the bound for the coefficients of the intermediate values. This gives us an opportunity to define a dual algorithm in the spectrum working properly with respect to a specific DFT map. We present these arguments formally in the next theorem.

Theorem 7.3. *Algorithm 7.3.4 computes an almost spectral reduction* $y(t) \equiv x^{-e}(t)$ *mods* $n(t)$ *such that the output signal* $y(t) \in \mathscr{B}_r^d$ *where* $r = b^2 d + b$. *Moreover, the coefficients of the intermediate values satisfy* $0 \leq y_i < u + b^2 d$ *for* $i = 0, 1, \ldots, d-1$.

Proof. First of all, the algorithm computes an evaluation polynomial $y(t)$ such that $y(t) \equiv x(t) \cdot t^{-e}$ mods $n(t)$. This can be seen as follows: The value of $y(t)$ is accumulated either by adding a multiple of $n(t)$ or a term shift (i.e., Steps 7 and 8). Obviously, either adding a multiple of $n(t)$ or dividing when the least significant coefficient is zero does not change the residue class that $y(t)$ belongs to. Therefore, the result naturally follows because of shifting exactly e times.

Now, lets examine how big the coefficients get: by Definition (7.4), $(x, x(t)) \in \mathscr{B}_u^e$ implies that $0 \leq x_i < u$ for all $i = 0, 1, \ldots, e-1$. Since $\deg(\underline{n}_d(t)) = d$ at every accumulation of the loop, $0 \leq y_i < u$ for $i \geq d$. In particular, at the last d run $y_i = 0$ for $i > d$ and $0 \leq y_d < b^2$.

On the other hand, when $i < d$ the coefficients of y_{d-i} satisfies

$$0 \leq y_{d-i} < \beta \cdot \underline{n}_d(i+1) + u < b^2(i+1) + u \,.$$

Observe that, a bound for y_{d-i} given by $0 \leq y_i < b^2 d + u$ for $i = 0, 1, \ldots, d-1$ is attained when $i = d-1$. The last d run is again special, since the reduction eliminates the u value from y_d at every accumulation. Therefore the final output satisfies

$$\begin{aligned} 0 \leq y_i &< (d-i)b^2 + b \\ &< r = b^2 d + b \quad \Longrightarrow \quad y(t) \in \mathscr{B}_r^d \end{aligned} \tag{7.6}$$

where the b factor comes from the last $\alpha(t)$ addition of Step 10 (note that, we assumed $deg(\alpha) < d-1$).

Note that Algorithm 7.3.4 describes a reduction routine of an arbitrary evaluation polynomial having degree $e-1$ larger than or equal to the degree of the modulus $d-1$. In fact, this does not really address the situation in a multiply–reduce methodology in which degrees are related. We give a corollary to Theorem 7.3 which cooperates with this situation.

Corollary 7.2. *Let $n(t)$ be a base b polynomial with degree $s-1$ such that $s = \lceil d/2 \rceil$ and let $x(t) \in \mathscr{B}_r^d$ where $r = sb^2$. Algorithm 7.3.4 computes an almost spectral reduction, $y(t) \equiv x(t) \cdot t^{-d}$ mods $n(t)$ in the polynomial frame $\mathscr{B}_{r'}^s$ where $r' = b^2 s + b$. Moreover, coefficients of all the intermediate values do not exceed $2b^2 s$.*

Proof. Let $n(t)$ be a base b polynomial with degree $s-1$ such that $s = \lceil d/2 \rceil$ and let $x(t) \in \mathscr{B}_r^d$ where $r = b^2 s$. Observe that the coefficients of $x(t)$ satisfy $0 \leq x_i < r = b^2 s$ for all $i = 0, 1, \ldots, d-1$ (note that we take $x(t)$ with the maximum degree $d-1$ in order to find the upper bounds). The algorithm drops the $\deg(x(t))$ to $\deg(n(t) = s-1$ and computes the almost spectral reduction of $x_{-d}(t) = x(t) \cdot t^{-d}$ in the frame $\mathscr{B}_{r'}^s$. The radius $r' = b^2 s + b$ can be deduced using Theorem 7.3. Moreover, since $0 \leq x_i < b^2 s$, the intermediate values are bounded by

$$0 \leq y_i < b^2 s + r = b^2 s + b^2 s = 2b^2 s.$$

7.3.5 Spectral Modular Reduction in a Finite Ring Spectrum

In this section, we translate the time simulation (i.e Algorithm 7.3.4) into the spectrum. We perform a line-by-line translation using the properties of DFT.

Our next step is to prove that Algorithm 7.3.4 and 7.3.5 agree; in other words there exists a DFT relation between the intermediate and output values in two domains at all times.

Theorem 7.4. *Algorithm 7.3.5 computes the almost spectral reduction, $Y(t) \equiv X^{-e}(t)$ mods $N(t)$ such that the inverse of the output signal $Y(t)$ gives $y(t) \equiv x^{-e}(t)$ mods $n(t)$ (i.e., the output of the Algorithm 7.3.4).*

Proof. Let $(x(t), X(t))$ and $(\underline{n}(t), \underline{N}(t))$ are transform pairs. In Step 4, we start with computing the last significant coefficient, y_0 of the time polynomial $y(t)$ using the

Spectral Reduction Algorithm (in a finite ring spectrum)
Suppose that there exists a DFT map $DFT_d^{\omega} : \mathscr{B}_q^e \rightarrow \mathscr{F}_q^e$ and

$$x(t) \xleftrightarrow{DFT} X(t), \qquad \underline{n}(t) \xleftrightarrow{DFT} \underline{N}(t)$$

where $(x(t), x) \in \mathscr{B}_r^e$ for $r < q - b^2 d$ and $(\underline{n}(t), \underline{n}) \in \mathscr{B}_b^{d+1}$ such that $\deg(n(t)) = d \leq e$ and $\underline{n}$ is a multiple of modulus n with $\underline{n}_0 = 1$ (we assume $\gcd(b, n) = 1$).
Input: $X(t)$ and $\underline{N}(t)$, spectral polynomials
Output: $Y(t) \equiv X^{-e}(t) \bmod N(t)$,

1: $Y(t) := X(t)$
2: $\alpha := 0$
3: **for** $i = 0$ **to** $e - 1$
4: $\quad y_0 := e^{-1} \cdot (Y_0 + Y_1 + \ldots + Y_d) \bmod q$
5: $\quad \beta := -(y_0 + \alpha) \bmod b$
6: $\quad \alpha := (y_0 + \alpha + \beta)$ **div** b
7: $\quad Y(t) := Y(t) + \beta \cdot \underline{N}(t) \bmod q$
8: $\quad Y(t) := Y(t) - (y_0 + \beta)(t) \bmod q$
9: $\quad Y(t) := Y(t) \odot \Gamma(t) \bmod q$
10: **end for**
11: $Y(t) := Y(t) + A(t)$,
12: **return** $Y(t)$

where $A(t)$ is the DFT pair of the base polynomial of α.

shifting property of DFT. Note that in Algorithm 7.3.4, y_0 comes for free. Once y_0 is computed, in Steps 5 and 6, the parameters β and the next carry α are generated.

In Step 7, a β multiple of $\underline{N}(t)$ is added to $Y(t)$, which updates $Y(t)$ such that $y_0 = 0 \bmod b$. By linearity, this step is equivalent to Step 6 of Algorithm 7.3.4.

Now, a division by t can be performed but before this shift, we need to eliminate the contribution of y_0 to the spectral polynomial $Y(t)$ completely. Since Step 7 updates y_0 to $y_0 + \beta$, the computation of $(Y(t) - (y_0 + \beta)(t))$ in Step 8 sets zeroth time term of $Y(t)$ to zero (observe that $(y_0 + \beta) \in \mathbb{Z}$ is a constant so $(y_0 + \beta)(t)$ is a fixed term polynomial, see Notation 2). If this is followed by the componentwise multiplication with $\Gamma(t)$ polynomial, Steps 8 and 9 together implement a logical circular shift (see Section 7.2.3).

Hence, we conclude that Algorithm 7.3.5 working in the spectrum agrees with Algorithm 7.3.4. However, we still need to find the domain for which these algorithms agree. We assume $\deg(x(t)) = e$ for $x(t) \in \mathscr{B}_r^e$ which implies that $0 \leq x_i < r = (q - b^2 d)$ for $i = 0, 1, \ldots, e - 1$. Since $\underline{n}$ is a multiple of modulus n with $\underline{n}_0 = 1$, we conclude by Theorem 7.3 that the intermediate values and the output $y(t)$ of the time simulation bounded by

$$0 \leq y_i < r + b^2 d = q - b^2 d + b^2 d = q$$

Therefore, no overflows occur, Algorithms 7.3.4 and 7.3.5 generate the transform pair $y(t)$ and $Y(t)$. As Algorithm 7.3.4 computes $y(t) \equiv x^{-e}(t) \bmod n(t)$, Algorithm 7.3.5 performs $Y(t) \equiv X^{-e}(t) \bmod N(t)$.

With Algorithm 7.3.5 we have completed our primary discussion on spectral modular reduction. We leave the improvement-related comments to Section 7.4. Notice that our presentation so far targets the reduction of an arbitrary evaluation polynomial of degree e with respect to a base polynomial of degree $d < e$. In the next section, we change this routine and target to reduce an evaluation polynomial which is a result of a convolution. After this, we introduce the spectral modular multiplication.

7.3.6 Spectral Modular Multiplication (SMM)

Convolution and the SMR algorithm can easily be combined to harvest a spectral modular multiplication algorithm in a finite ring spectrum. In order to have a clear presentation we divide our presentation into 3 subprocedures as seen in Figure 7.3. Note that the initial and final stages consist of some data arrangements where the *Spectral Modular Product* (SMP) procedure consists of the actual multiplication and reduction steps (i.e., convolution and spectral reduction). Later, while presenting the spectral exponentiation algorithm, SMP is going to be the basic building block again. The SMP procedure and SMM are given as follows:

Since we operate in a finite ring spectrum, once again we need to deal with the overflows that might occur during the computations. In fact, Algorithm 7.3.6 gives a correct result if the intermediate values stay bounded. As a next step, we state the condition when overflows do not occur starting with two lemmas.

Lemma 7.3. *SMP*$(X^d(t), Y(t)) \equiv X(t) \odot Y(t)$ *mods* $N(t)$

Proof. Since SMP$(X^d(t), Y(t))$ computes the spectral coefficients of almost modular reduction, $Z(t) \equiv (X^d(t) \odot Y^{-d}(t))$ mods $N(t)$, hence taking the inverse transform gives

$$x^d(t)y^{-d}(t) = x(t) \cdot t^d y(t) \cdot t^{-d} = x(t) \cdot y(t) \text{ mods } n(t) \ .$$

Lemma 7.4. *Procedure 7.3.6 computes* $Z(t) \equiv (X(t) \odot Y^{-d}(t))$ *mods* $N(t)$ *if the parameters* b, q *and* s *satisfies the following inequality*

$$2sb^2 < q \tag{7.7}$$

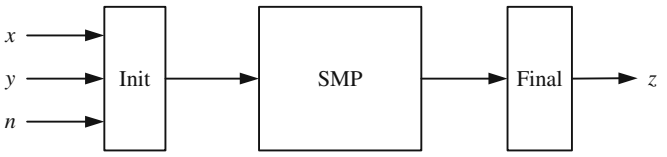

Fig. 7.3 Spectral Modular Multiplication.

Spectral Modular Product
Suppose that there exists a DFT map $DFT_d^{\omega} : \mathscr{B}_q^d \rightarrow \mathscr{F}_q^d$, and $X(t), Y(t)$ and $\underline{N}(t)$ be transform pairs of $x(t), y(t)$ and $\underline{n}(t)$ respectively where $(x(t), x)$ and $(y(t), y)$ are evaluation polynomials in the frame $\mathscr{B}_r^s$ with $r > 0$ and $s = \lceil d/2 \rceil$, and $(\underline{n}(t), \underline{n}) \in \mathscr{B}_b^{s+1}$ such that $\deg(\underline{n}(t)) \leq s$ and $\underline{n}$ is a multiple of modulus n with $\underline{n}_0 = 1$ (we assume $\gcd(b, n) = 1$).
Input: $X(t), Y(t)$ and $\underline{N}(t)$; spectral polynomials
Output: $Z(t) \equiv (X(t) \odot Y^{-d}(t)) \bmod N(t)$,
procedure SMP$(X(t), Y(t))$

1: $Z(t) := X(t) \odot Y(t)$
2: $\alpha := 0$
3: **for** $i = 0$ **to** $d - 1$
4: $\quad z_0 := d^{-1} \cdot (Z_0 + Z_1 + \ldots + Z_d) \bmod q$
5: $\quad \beta := -(z_0 + \alpha) \bmod b$
6: $\quad \alpha := (z_0 + \alpha + \beta)/b$
7: $\quad Z(t) := Z(t) + \beta \cdot \underline{N}(t) \bmod q$
8: $\quad Z(t) := Z(t) - (z_0 + \beta)(t) \bmod q$
9: $\quad Z(t) := Z(t) \odot \Gamma(t) \bmod q$
10: **end for**
11: $Z(t) := Z(t) + \alpha(t)$
12: **return** $Z(t)$

Spectral Modular Multiplication
Suppose that there exists a DFT map $DFT_d^{\omega} : \mathscr{B}_q^d \rightarrow \mathscr{F}_q^d$. Let $n(t)$ be a base b polynomial for n where $\deg(n(t)) = s - 1$, $s = \lceil d/2 \rceil$ and $\gcd(b, n) = 1$.
Input: A modulus $n > 0$ and $x, y < n$
Output: An almost modular reduction $z \equiv xy \bmod n$

1: Compute $\underline{n} = \delta \cdot n$ such that the base polynomial
$\underline{n}(t)$ has degree d and $\underline{n}_0 = 1$
2: $\underline{N}(t) := \mathrm{DFT}_d^{\omega}(\underline{n}(t))$
3: Compute the base polynomial $\lambda(t)$, $\lambda = b^d \bmod n$.
4: Compute the base polynomial $x^d(t) = x(t) \cdot t^d$
for $x \cdot \lambda \bmod n$.
5: $X^d(t) := \mathrm{DFT}_d^{\omega}(x^d(t))$
6: $Y(t) := \mathrm{DFT}_d^{\omega}(y(t))$
7: $Z(t) := \mathrm{SMP}(X^d(t), Y(t))$
8: $z(t) := \mathrm{IDFT}_d^{\omega}(Z(t))$
9: **return** $z(b)$

Proof. In the previous sections, we described the action of the convolution and how the steps of reduction work. Here, we concentrate on driving the Inequality (7.7). Assuming that the conditions of SMP are satisfied, we investigate the time simulation of the algorithm in order to trace the overflows.

Using Theorem 7.3, observe that after convolution at Step 1, the time polynomial $z(t)$ doubles its degree to $2s - 2$. Moreover, the magnitude of its coefficients cannot exceed sb^2 since $x(t)$ and $y(t)$ are base b polynomials (i.e., $z(t) \in \mathscr{B}_r^d$ where $r = sb^2$).

When it comes to analyzing the reduction steps, applying Corollary 7.2 assures that the output $z^{-d}(t)$ belongs to $\mathscr{B}_{r'}^{d}$ where $r' = b^2s + b$ and coefficients of all the intermediate values do not exceed $2b^2s$. Therefore, if q is chosen as $\max(2b^2s, b^2s + b) = 2b^2s < q$, no overflow is generated and SMP computes the desired result. Note here that the carry added in Step 12 is no longer large because of working with a convolution output, hence we take it as a constant rather than breaking it into words.

Theorem 7.5. *Algorithm 7.3.6 computes an almost modular reduction,* $z \equiv xy \bmod n$, *if the parameters* b, q *and* s *satisfy* $2sb^2 < q$.

Proof. Notice that in the initialization steps of Algorithm 7.3.6, before calculating the Fourier coefficients, we compute $xb^d \bmod n$ (i.e., $x^d(t) \bmod n(t)$). Using Lemma 7.3, one can see that Step 7 computes the product $x(t)y(t)$ mods $n(t)$ unless overflows occur.

Since the initialization and finalization parts do not change the coefficient bounds, one can get the minimal domain for the core SMP as $2sb^2 < q$ using Lemma 7.4.

7.3.7 Spectral Modular Exponentiation

In general, a single classical modular multiplication is faster than a single SMM; however, spectral methods are very effective when several modular multiplications with respect to the same modulus are needed. An example is the case when one needs to compute a modular exponentiation, i.e., the computation of $m^e \bmod n$, where m, e and n are positive integers. Such a setup needs a consecutive use of SMM; it also means a consecutive use of DFT and IDFT operations (obviously redundant computations as seen in Figure 7.4). Therefore, if the data is kept in the Fourier domain at all times, the backward and forward transforms are bypassed. Consequently,

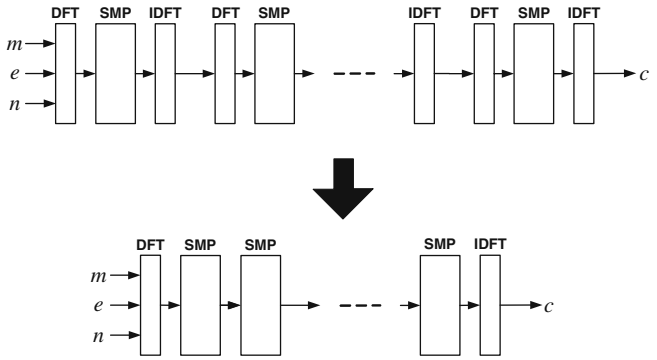

Fig. 7.4 Spectral Modular Exponentiation.

this approach decreases the asymptotic crossovers of the spectral methods to cryptographic sizes while computing the modular exponentiation.

There are many methods for carrying out general exponentiation. Mostly, efficiency comes from two resources; one is to decrease the time to multiply; the other is to reduce the number of multiplications. In practice one does both. Notice that, until now our objective was reducing the modular multiplication which is categorized in the first category. For the rest of this study, we keep this goal and simply consider using the binary method (see [8]) for the rest of our presentation.

The binary method scans the bits of the exponent either from left to right or from right to left. A squaring is performed at each step, and depending on the scanned bit value, a subsequent multiplication is performed. We describe the spectral modular exponentiation algorithm by using a left-to-right binary method below.

Remark 7.5. Since the SME algorithm computes an almost modular reduction of $c \equiv m^e \bmod n$, a final reduction may be needed if the output is desired in the range $[0, n-1]$.

Spectral Modular Exponentiation
Suppose that there exists a DFT map $DFT_d^\omega : \mathscr{B}_q^d \rightarrow \mathscr{F}_q^d$. Let $n(t)$ be a base b polynomial for n where $\deg(n(t)) = s-1$, $s = \lceil d/2 \rceil$ and $\gcd(b,n) = 1$.
Input: A modulus $n > 0$ and $m, e < n$
Output: An almost modular reduction, $c \equiv m^e \bmod n$.

1: Compute $\underline{n} = \delta \cdot n$ such that the base polynomial $\underline{n}(t)$ has degree d and $\underline{n}_0 = 1$
2: $\underline{N}(t) := \mathrm{DFT}_d^\omega(\underline{n}(t))$
3: Compute the base b polynomial $(\lambda', \lambda'(t))$ where $\lambda' = b^{2d} \bmod n$.
4: $\Lambda'(t) := \mathrm{DFT}_d^\omega(\lambda'(t))$
5: $M(t) := \mathrm{DFT}_d^\omega(m(t))$
6: $\overline{M}(t) := \mathrm{SMP}(M(t), \Lambda'(t))$
7: $\overline{C}(t) := \mathrm{SMP}(1(t), \Lambda'(t))$
8: **for** $i = j-2$ **downto** 0
9: $\quad \overline{C}(t) := \mathrm{SMP}(\overline{C}(t), \overline{C}(t))$
10: $\quad$ **if** $e_i = 1$ **then** $\overline{C}(t) := \mathrm{SMP}(\overline{C}(t), \overline{M}(t))$
11: $C(t) := \mathrm{SMP}(\overline{C}(t), 1(t))$
12: $c(t) := \mathrm{IDFT}_d^\omega(C(t))$
13: **return** $c(b)$

Once again, we need to guarantee that overflows do not occur; in other words, the coefficients of intermediate or final results should not be winding over q. We start with a lemma, stating how big the coefficients of a special polynomial get after a convolution, then, using this result we comment on how q has to be chosen to avoid overflows.

Lemma 7.5. *Let $s > 0$ and $x(t) = 1 + 2t + 3t^3 + \ldots + st^{s-1}$, then the coefficients of $z(t) = (x(t))^2$ are bounded by*

$$B(s) = \frac{-2s^3}{3} - s^2 + 2s^2 r_1 - \frac{s}{3} + \frac{r_1}{3} + 2sr_1 \tag{7.8}$$

where $r_1 = -2 + \sqrt{\frac{3+18s^2+18s}{9}}$.

Proof. Let $x(t) = 1 + 2t + 3t^3 + \ldots + st^{s-1}$ be a polynomial of degree $s-1$. Observe that if the convolution $z(t) = (x(t))^2$ is computed, the coefficients of $z(t)$ satisfies the following recurrence:

$$\begin{aligned}
z_0 &= 1 \\
z_1 &= 2^2 \\
z_2 &= 3^2 + z_0 \\
&\vdots \\
z_{s-1} &= s^2 + z_{s-3} \\
\\
z_s &= (s-1)(s+1) + z_{s-2} \\
z_{s+1} &= 2(s-2)(s+1) + z_{s-3} \\
&\vdots \\
z_{s+i-1} &= i(s-i)(s+1) + z_{s-i-1} \\
&\vdots \\
z_{2s-2} &= (s-1)(s+1) + z_0
\end{aligned}$$

If these coefficients are examined carefully one realizes that the coefficients up to the $(s-2)$th are monotonously increasing and $z_s > z_{s-2}$ for $s > 1$. Therefore, a maximum magnitude which also decides the bound has to be located somewhere in between coefficients $s-1$ and $2s-2$.

Observe that z_i is a telescoping sequence for $r < s$, in other words,

$$z_{r+1} + z_r = 1^2 + 2^2 + \ldots + (r+2)^2 = \sum_{i=1}^{r+2} i^2 \ .$$

This sum can be written as

$$z_{r+1} + z_r = \frac{(B + (r+2) + 1)^3 - B^3}{3} \tag{7.9}$$

where B^i stands for the ith Bernolli number (i.e., $B^0 = B_0 = 1, B_1 = -1/2, B_2 = 1/6$ and $B_3 = 1/30$. Plugging Bernolli numbers to the Equation (7.9) gives

$$z_{r+1} + z_r = \frac{1}{6}(2r^3 + 15r^2 + 37r + 30).$$

Here, if $r+1$ is even, then z_{r+1} and z_r can be found as

$$z_r = z_1 + z_3 + \ldots + z_r = \frac{1}{6}(r^3 + 6r^2 + 11r + 6)$$
$$z_{r+1} = \frac{1}{6}(r^3 + 9r^2 + 26r + 24)$$

and, in case $r+1$ is odd, one would get

$$z_{r+1} = z_1 + z_3 + \ldots + z_{r+1} = \frac{1}{6}(r^3 + 9r^2 + 26r + 24)$$
$$z_r = \frac{1}{6}(r^3 + 6r^2 + 11r + 6).$$

Therefore, in either case, z_r can be written as $\frac{1}{6}(r^3 + 6r^2 + 11r + 6)$ and a general term of the recurrence would be found by plugging z_r into the system after replacing the index $s+i-1$ by r,

$$z_r = \begin{cases} \frac{1}{6}(r^3 + 6r^2 + 11r + 6) & \text{if } r < s \\ -\frac{2s^3}{3} + s^2 r + \frac{5s}{3} - \\ \frac{r^3}{6} - \frac{11r}{6} + s^2 + sr - r^2 - 1 & \text{if } s \le r < 2s-1 \end{cases}$$

In order to find the maximum value of the z_r function, we substitute the roots of the derivative $\frac{\partial z_r}{\partial r}$ into the equation of z_r. Observe that the root $r_1 = -2 + \sqrt{\frac{3+18s^2+18s}{9}}$ gives the local maximum in the range $s \le r < 2s-1$ and if r_1 is plugged into z_r, one would get the bound $B(s)$ as a function of s

$$B(s) = \frac{-2s^3}{3} - s^2 + 2s^2 r_1 - \frac{s}{3} + \frac{r_1}{3} + 2sr_1,$$

which in fact gives the desired bound.

Theorem 7.6. *Algorithm 7.3.7 computes* $c \equiv m^e \bmod n$ *if the parameters* b, q *and* s *satisfy the following inequality*

$$(b^2 + b)^2 B(s) + b^2 s < q \tag{7.10}$$

where $B(s)$ *is given by Equation (7.8).*

Proof. First of all, SME implements the binary exponentiation method with a Montgomery type multiplier SMP all working in the spectrum. Thus, the algorithm works as long as a nice domain is chosen for all intermediate values and output causing no overflows. Recall by Theorem 7.3 that if the inputs of SMP are spectral base b polynomials, the output of SMP algorithm is a spectral polynomial having a time pair fitting into the frame $\mathscr{B}_r^s$ with $r = b^2 s + b$. However, in the case of a consecutive SMP usage, the output of the second SMP would have larger time coefficients. For

instance, in Steps 9, 10 and 11 the input $\overline{C}(t)$ is a spectral polynomial with time coefficients larger than b. If these steps are examined further, one understands that maximum magnitudes are attained from the computation of SMP$(\overline{C}(t),\overline{C}(t))$ in Step 9, since for both Steps 10 and 11, $\overline{M}(t)$ and $1(t)$ are spectral base b polynomials.

Now, we investigate how big the coefficients of the time polynomial get after Step 9. In order to have a better analysis, we look at the distribution of the coefficients of the time polynomial after applying SMP. In Theorem 7.3, we showed that after a reduction $c(t) \in \mathscr{B}_r^s$ has the form

$$c(t) = c_0 + c_1 t + c_2 t^2 + \ldots + c_{s-1} t^{s-1},$$

where $c_i < (s-i)b^2 + b$ for $i = 0, 1, \ldots, s-1$. Since $c_i < (s-i)b^2 + b < (s-i)(b^2 + b)$, we write

$$c(t) < (b^2 + b)y(t),$$

where $y(t) = s + (s-1)t + (s-2)t^2 + \ldots + 1t^{s-1}$. If $(c(t))^2$ is computed as seen in Step 9, we have

$$(c(t))^2 < (b^2 + b)^2 (y(t))^2,$$

and using Lemma 7.5, we guarantee that the coefficients of $(y(t))^2$ are bounded by $B(s)$ [1], which implies that $(b^2 s + b)^2 B(s)$ bounds the coefficients of $(c(t))^2$. When it comes to the intermediate values, because of the reduction steps, coefficients get slightly larger which is given by Theorem 7.3 as $(b^2 + b)^2 B(s) + b^2 s$. Therefore, if q is chosen larger than this final bound, no overflow is generated and the DFT respects SME over the ring $\mathbb{Z}_q$.

Inequality (7.10) is very centric as it gives the relation between the parameters b, s and q in a consecutive use of SMP algorithm. In practice, these parameters are generated in two different ways: the first one is picking s and b and then finding a suitable ring $\mathbb{Z}_q$ that admits a DFT of size d, while the second one is picking a ring with q elements, decide on s, and then find out the base b. We discuss the parameter selection-related issues in the next chapter after giving an illustrative example.

We conclude that $q > 131845.0 > 2^{17}$. Thus we need to search for a Fermat or Mersenne ring with $q \geq 2^{18} - 1$ that admits a DFT with length $d = 7$ or $d = 8$ for ω equal to a power of two. It turns out that the ring $\mathbb{Z}_{2^{20}+1}$ satisfies these conditions with $\omega = 32$.

7.3.8 Illustrative Example

In this section, we present the temporary values and the final result of an exponentiation computation (i.e., $c = m^e \pmod{n}$) using the SME method with the input values as $m = 2718$, $e = 53$, and $n = 3141$.

[1] $c(t)$ of Lemma 7.5 is the mirror image of $y(t)$

If we select the parameters $b = 2^3$ and $s = 4$, Inequality (7.10) assures that SME works correctly in a ring having $q > 131845$ elements. In order to have some computational convenience, we chose the Fermat ring $\mathbb{Z}_{2^{20}+1}$ which admits a DFT with length $d = 7$ for $\omega = 32$.

With these selections we compute $d^{-1} \bmod q$ and $\Gamma(t)$ as

$$d^{-1} = 8^{-1} \pmod{2^{20}+1} = 917505 .$$

and

$$\begin{aligned}\Gamma(t) &= 1 + w^{-1}t + w^{-2}t^2 + w^{-3}t^3 + w^{-4}t^4 + w^{-5}t^5 + w^{-6}t^6 + w^{-7}t^7 \\ &= 1 + 1015809t + 1047553t^2 + 1048545t^3 + 1048576t^4 + \\ &\quad 32768t^5 + 1024t^6 + 32t^7 .\end{aligned}$$

We start with writing m and n in polynomial representation

$$\begin{aligned}n(t) &= 5 + 0 \cdot t + 1t^2 + 6t^3 , \\ m(t) &= 6 + 3t + 2t^2 + 5t^3 .\end{aligned}$$

Note that $\deg n(t) = s - 1 = 3$ and $\gcd(n, b) = 1$.

The steps of the SME method computing this modular exponentiation are described below.

1. Given $n = 3141$, we have $n_0 = 5$. Finding the inverse of n_0 modulo b gives δ which is mostly achieved by Extended Euclidean algorithm:

   $$\delta = n_0^{-1} \bmod b = 5 \bmod 8 .$$

   Thus, $\underline{n} = \delta n = 5 \cdot 3141 = 15705$ which is equal to

   $$\underline{n}(t) = 1 + 3t + 5t^2 + 6t^3 + 3t^4$$

   in polynomial representation. Recall that $\deg(\underline{n}(t)) = s = 4$ and $\underline{n}_0 = 1$
2. The computation of $\underline{n}(t) = \mathrm{DFT}[\underline{n}(t)]$ can be accomplished by a matrix multiplication or, for more efficiency, some FFT can be employed. We obtain the result of the DFT as

   $$\underline{N}(t) = 18 + 201822t + 1045504t^2 + 93374t^3 + 856991t^5 + 3071t^6 + 944959t^7$$

   Recall that we work in the finite ring $\mathbb{Z}_q$ with $q = 2^{20} + 1 = 1048577$ represented by the least residue set; thus, the coefficients of the polynomial $\underline{N}(t)$ are in the range $[0, 2^{20})$.
3. After computing $\lambda' = 2^{2d} \bmod n = 8^{16} \bmod 3141 = 415$, the polynomial representation of λ' is found

   $$\lambda'(t) = 7 + 3t + 6t^2 .$$

Furthermore, we obtain the spectral coefficients of $\Lambda'(t)$ using the DFT as

$$\Lambda'(t) = 16 + 6247t + 3073t^2 + 92167t^3 + 10t^4 + 6055t^5 + 1045506t^6 + 944136t^7$$

4. Given $m(t)$, we obtain its spectral representation $M(t)$ using the DFT as

$$M(t) = 16 + 165990t + 1046533t^2 + 96422t^3 + 886695t^5 + 2052t^6 + 948071t^7$$

5. The SMP algorithm is used to compute $\overline{M}(t) = \text{SMP}[M(t), \Lambda'(t)]$ with inputs

$$\begin{aligned} M(t) &= 16 + 165990t + 1046533t^2 + 96422t^3 + 886695t^5 + 2052t^6 + 948071t^7 \\ \Lambda'(t) &= 16 + 6247t + 3073t^2 + 92167t^3 + 10t^4 + 6055t^5 + 1045506t^6 + 944136t^7 \end{aligned}$$

We then use the SMP to find the resulting polynomial $\overline{M}(t)$ given the inputs $M(t)$ and $\Lambda'(t)$. First we execute Step 1 in the SMP method, and obtain the initial value of $Z(t)$ using the rule $Z_i = M_i \cdot \Lambda'_i \bmod q$ for $i = 0, 1, \ldots, 7$ as

$$Z(t) = 256 + 945454t + 10250t^2 + 236399t^3 + 223985t^5 + 1038347t^6 + 691376t^7$$

In Step 2 of the SMP method, we assign the initial value of $\alpha = 0$, and start the loop for $i = 0, 1, \ldots, 7$. We illustrate the computation of the instance of the loop for $i = 0$ in Table 7.1. The **for** loop needs to execute the remaining values of i as $i = 1, 2, \ldots, 7$ in order to compute the resulting product $\overline{M}(t)$ given by

$$\begin{aligned} \overline{M}(t) = {} & 135 + 324054t + 36891t^2 + 398677t^3 + 27t^4 + \\ & 779927t^5 + 1011740t^6 + 594712t^7 . \end{aligned}$$

Table 7.1 The SMP **for** loop instance $i = 0$.

Step	Operation and Result
4:	$z_0 = d^{-1} \cdot (Z_0 + Z_1 + Z_2 + Z_3 + Z_4 + Z_5 + Z_6 + Z_7) \pmod q$ $z_0 = 917505 \cdot (256 + 945454 + 10250 + 236399 + 223985 +$ $1038347 + 691376) \pmod{1048567} = 42$
5:	$\beta = -(z_0 + \alpha) \pmod b = -(42 + 0) \pmod{16} = 6$
6:	$\alpha = (z_0 + 0 + \beta)/b = (42 + 6)/16 = 3$
7:	$Z_i = Z_i + \beta \cdot \overline{N}_i \pmod q$ $Z(t) = 364 + 59232t + 1040389t^2 + 796643t^3 +$ $123046t^5 + 8196t^6 + 69668t^7$
8:	$Z_i = Z_i - (z_0 + \beta) = Z_i - 48 \pmod q$ $Z(t) = 316 + 59184t + 1040341t^2 + 796595t^3 + 1048529t^4 +$ $122998t^5 + 8148t^6 + 69620t^7$
9:	$Z_i = Z_i \cdot \Gamma_i \pmod q$ $Z(t) = 316 + 526138t + 45048t^2 + 723385t^3 + 48t^4 +$ $717053t^5 + 1003513t^6 + 130686t^7$

6. In this step, the SMP method is used to compute $\overline{C}(t) = \text{SMP}[1(t), \Lambda'(t)]$ with inputs

$$\begin{aligned} 1(t) &= 1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 \,, \\ \Lambda'(t) &= 16 + 6247t + 3073t^2 + 92167t^3 + 10t^4 + 6055t^5 + 1045506t^6 + 944136t^7 \end{aligned}$$

We will not give the details of this multiplication since it is similar to the previous one. The result is obtained as

$$\begin{aligned} \overline{C}(t) = {} & 106 + 13591t + 39979t^2 + 217142t^3 + 28t^4 + \\ & 11095t^5 + 1008684t^6 + 806969t^7 \,. \end{aligned}$$

7. **Exponentiation Loop:** The loop starts with the values of $\overline{M}(t)$ and $\overline{C}(t)$ computed above as

$$\begin{aligned} \overline{M}(t) = {} & 135 + 324054t + 36891t^2 + 398677t^3 + 27t^4 + \\ & 779927t^5 + 1011740t^6 + 594712t^7 \,, \\ \overline{C}(t) = {} & 106 + 13591t + 39979t^2 + 217142t^3 + 28t^4 + \\ & 11095t^5 + 1008684t^6 + 806969t^7 \,. \end{aligned}$$

Given the exponent value $e = (53)_{10} = (110101)_2$, the exponentiation algorithm performs squarings and multiplications using the SMP method. Since $j = 6$, the value of i starts from $i = 5$ and moves down to zero, and computes the new value of $\overline{C}(t)$ using the binary method of exponentiation as described. The steps of the exponentiation and intermediate values of $\overline{C}(t)$ are tabulated in Table 7.2. The final value is computed as

$$\begin{aligned} \overline{C}(t) = {} & 174 + 327348t + 43062t^2 + 592243t^3 + 54t^4 + \\ & 782837t^5 + 1005623t^6 + 395062t^7 \,. \end{aligned}$$

8. After the exponentiation loop is completed, we have the final value $\overline{C}(t)$. In this step, we have an SMP execution followed by an inverse DFT calculation.
9. We obtain $C(t)$ from $C(t) = \text{SMP}[\overline{C}(t), 1(t)]$ using the inputs

$$\begin{aligned} \overline{C}(t) = {} & 174 + 327348t + 43062t^2 + 592243t^3 + 54t^4 + \\ & 782837t^5 + 1005623t^6 + 395062t^7 \,. \\ 1(t) = {} & 1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 \,. \end{aligned}$$

This computation finds $C(t)$ as

$$\begin{aligned} C(t) = {} & 169 + 438168t + 48142t^2 + 842167t^3 + 27t^4 + \\ & 696537t^5 + 1000463t^6 + 120506t^7 \,, \end{aligned}$$

Table 7.2 The steps of the exponentiation loop.

i	e_i	Operation	$\overline{C}(t)$
		Start	$106+13591t+39979t^2+217142t^3+$ $28t^4+11095t^5+1008684t^6+806969t^7$
5		$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{C}(t)]$	$127+13519t+36931t^2+118862t^3+$ $55t^4+11215t^5+1011780t^6+905297t^7$
	1	$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{M}(t)]$	$135+324054t+36891t^2+398677t^3+$ $27t^4+779927t^5+1011740t^6+594712t^7$
4		$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{C}(t)]$	$127+118020t+35890t^2+178339t^3+$ $45t^4+967557t^5+1012787t^6+833510t^7$
	1	$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{M}(t)]$	$175+434919t+42016t^2+648646t^3+$ $45t^4+693672t^5+1006625t^6+320201t^7$
3		$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{C}(t)]$	$119+526344t+16391t^2+982536t^3+$ $27t^4+589897t^5+1032200t^6+1047114t^7$
	0		$119+526344t+16391t^2+982536t^3+$ $27t^4+589897t^5+1032200t^6+1047114t^7$
2		$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{C}(t)]$	$202+128046t+60499t^2+955597t^3+$ $72t^4+976047t^5+988244t^6+37904t^7$
	1	$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{M}(t)]$	$192+628407t+33843t^2+586390t^3+$ $54t^4+494072t^5+1014836t^6+388633t^7$
1		$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{C}(t)]$	$265+755301t+60454t^2+460547t^3+$ $63t^4+422502t^5+988199t^6+459208t^7$
	0		$265+755301t+60454t^2+460547t^3+$ $63t^4+422502t^5+988199t^6+459208t^7$
0		$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{C}(t)]$	$296+546702t+74843t^2+734828t^3+$ $90t^4+606607t^5+973916t^6+209585t^7$
	1	$\overline{C}(t)=\text{SMP}[\overline{C}(t),\overline{M}(t)]$	$174+327348t+43062t^2+592243t^3+$ $54t^4+782837t^5+1005623t^6+395062t^7$

10. We obtain $c(t)$ using the inverse DFT function $c(t)=\text{IDFT}[C(t)]$, which gives

$$c(t)=56+59t+42t^2+12t^3\ .$$

Thus, the final value becomes $c(b)=9360\equiv 3078 \bmod 3141$, which is equal to

$$3078=22718^{53} \bmod 3141$$

as required.

7.4 Applications to Cryptography

Modular exponentiation is one of the most important arithmetic operation in modern cryptography. For example, the RSA algorithm requires exponentiation in $\mathbb{Z}_n$ for some positive integer n, whereas Diffie-Hellman key agreement and the ElGamal scheme use exponentiation in some large prime fields (see [9]).

In this chapter, we describe the methodologies of selecting the parameters for SME in order to use the method in public-key cryptography. We carefully investigate suitable rings and structures that makes spectral techniques available for modular arithmetic. In particular, the Inequality (7.10) presents a solid basis for the relation between the parameters b, q and s.

7.4.1 Mersenne and Fermat rings

An integer ring for which q is of the form $2^v \pm 1$ is the most suitable for the SME computation since the modular arithmetic operations for such q are simplified. Moreover, if the principal root of unity is chosen as a power of 2, spectral coefficients are computed only using additions and circular shifts. The rings of the form $2^v - 1$ are called the *Mersenne rings*, while the rings of the form $2^v + 1$ are called the *Fermat rings*. In Table 7.3, we tabulate some suitable Fermat and Mersenne rings for SME function. Furthermore, we also tabulate a root of unity and the DFT length for each ring.

Observe that, it is possible to attain larger transform lengths; however, such a principal root of unity brings some further complexity to the computations. To be specific, multiplications with roots of unity involve additions as well as cyclic shifts. Some cases such as $\omega = \pm\sqrt{2}$ can be tolerable for longer transform sizes but other choices could be very costly. For instance, in $\mathbb{Z}_{2^{20}+1}$, $\omega = 4100$ is not a power of 2, hence every single multiplication with roots of unity is a 20-bit by 20-bit multiplication and not tolerable for our purposes.

Table 7.3 Parameters of NTT for $2^{16} < q < 2^{81}$.

ring $\mathbb{Z}_q$	prime factors	(ω, NTT length)	
$2^{16}+1$	65537	$(4,16)$	$(2,32)$
$2^{17}-1$	131071	$(2,17)$	$(-2,34)$
$2^{19}-1$	524287	$(2,19)$	$(-2,38)$
$2^{20}+1$	$17 \cdot 61681$	$(32,8)$	$(4100,16)$
$2^{23}-1$	$47 \cdot 178481$	$(2,23)$	$(-2,46)$
$2^{24}+1$	$97 \cdot 257 \cdot 673$	$(8,16)$	$(\sqrt{8},32)$
$2^{29}-1$	$233 \cdot 1103 \cdot 2089$	$(2,29)$	$(-2,58)$
$2^{31}-1$	2147483647	$(2,31)$	$(-2,62)$
$2^{32}+1$	$641 \cdot 6700417$	$(4,32)$	$(2,64)$
$2^{37}-1$	$223 \cdot 616318177$	$(2,37)$	$(-2,74)$
$2^{40}+1$	$257 \cdot 4278255361$	$(32,16)$	$(\sqrt{32},32)$
$2^{41}-1$	$13367 \cdot 164511353$	$(2,41)$	$(-2,82)$
$2^{64}+1$	$274177 \cdot 67280421310721$	$(4,64)$	$(2,128)$
$2^{79}-1$	$2687 \cdot 202029703 \cdot 1113491139767$	$(2,79)$	$(-2,158)$
$2^{80}+1$	$414721 \cdot 44479210368001$	$(32,32)$	$(\sqrt{32},64)$

In general, the transform lengths tabulated above are considered too short for most of the digital signal processing applications. On the other hand, these lengths seem reasonable for cryptographic applications and our purposes.

7.4.2 Pseudo Number Transforms

The Mersenne and Fermat rings are not the only suitable rings for efficient arithmetic, if m (not necessarily a prime) is a small divisor of n. The rings of the form $Z_{n/m}$ are also quite useful.

Definition 7.17. Let n and m be positive integers and m divides n. The NTT defined over $\mathbb{Z}_{n/m}$ is called a **pseudonumber transform (PNT)** .

In general, arithmetic in $\mathbb{Z}_{n/m}$ is difficult; however, since m is a factor of n, the arithmetic modulo (n/m) can be carried in the ring Z_n. By selecting Z_n as a Mersenne or Fermat ring one simplifies the overall arithmetic. The next theorem makes the importance of PNT more clear.

Theorem 7.7. *Let $n = n_1 n_2 \dots n_l = m_1^{e_1} m_2^{e_2} \dots m_l^{e_l}$, where $n_i = m_i^{e_i}$ for $i = 1,2,\dots,l$ for distinct primes m_i and positive integers e_i and l. Let R be a proper subset of the set $\{n_1, n_2, \dots, n_l\}$ and $R' = \{m_i - 1 : m_i^{e_i} \in R\}$. If $S = \{m_1 - 1, m_2 - 1, \dots, m_l - 1\}$, then gcd(S) $\leq$ gcd(R') $=: d'$ and a PNT of length-d' can be defined over $\mathbb{Z}_{n/m}$ for $m = \prod_{n_i \notin R} n_i$.*

Proof. First, $R \subsetneq S \Rightarrow \gcd(S) \leq \gcd(R')$. For the second part, let R be a proper subset of the set $\{n_1, n_2, \dots, n_l\}$ such that $n/m = \prod_{n_i \in R} n_i$. Using Corollary 7.1, there exists an NTT with length $d' = \gcd(\{m_i - 1 : m_i^{e_i} \in R\})$ over $\mathbb{Z}_{n/m}$.

Example 7.6. In $\mathbb{Z}_{2^{15}-1}$, Corollary 7.1 states that the maximum transform length is $\gcd(6, 30, 150) = 6$. This MNT length is very short if the size of the ring is considered. On the other hand, if a PNT is employed in the ring $\mathbb{Z}_{(2^{15}-1)/7}$, we get the transform lengths up to $\gcd(30, 150) = 30$.

At first glance, the arithmetic in the ring $\mathbb{Z}_{(2^{15}-1)/7}$ seems difficult; however, it is possible to perform the actual computation in the ring $\mathbb{Z}_{(2^{15}-1)}$ with a final reduction to modulo $(2^{15} - 1)/7$.

Remark 7.6. Observe that PNT tailors the rings in a way that larger length transforms are possible. But while doing that, the size of the ring shrinks. The most interesting PNTs are the ones which enlarge the lengths with minimal shrinkage. The effective size of the decreased ring has to be concerned when PNTs are used.

In Table 7.4, we present parameters of some suitable pseudo-Mersenne and Fermat rings. If Tables 7.3 and 7.4 are combined, it is seen that for almost every v (recall that $n = 2^v \pm 1$) in between 16 and 41 there exists some sets of parameters for a nice NTT. Therefore, PNTs enrich the possible design choices which equip us to meet the marginal needs of particular applications.

Table 7.4 Suitable NTTs with ω and d values, $\star$ shows that n/m is a prime.

Ring n	PNT Modulus n/m	ω	d	ω	d
$2^{17}+1$	$(2^{17}+1)/3^\star$	$-2,4$	17	2	34
$2^{20}+1$	$(2^{20}+1)/17^\star$	4	20	2	40
$2^{23}-1$	$(2^{23}-1)/47^\star$	2	23	-2	46
$2^{23}+1$	$(2^{23}+1)/3^\star$	$-2,4$	23	2	46
$2^{25}-1$	$(2^{25}-1)/31$	2	25	-2	50
$2^{27}-1$	$(2^{27}-1)/511$	2	27	-2	54
$2^{28}+1$	$(2^{28}+1)/17^\star$	4	28	2	56
$2^{29}+1$	$(2^{29}+1)/3$	$-2,4$	29	2	58
$2^{31}+1$	$(2^{31}+1)/3^\star$	$-2,4$	31	2	62
$2^{34}-1$	$(2^{34}-1)/3$	2	34	-2	68
$2^{34}+1$	$(2^{34}+1)/5$	4	34	2	68
$2^{37}+1$	$(2^{37}+1)/3$	$-2,4$	37	2	74
$2^{39}-1$	$(2^{39}-1)/7$	2	39	-2	78
$2^{39}+1$	$(2^{39}+1)/9$	$-2,4$	39	2	78

7.4.3 Parameter Selection for RSA

In this section, we tabulate some SME parameters for modular exponentiation calculation suitable for RSA cryptosystems. Once the underlying ring, the DFT length and the principal root of unity are selected, the maximum modulus size used in the SME method is computed by finding the base $b = 2^u$. The relation between these parameters is computed after determining the maximum b satisfying the Inequality (7.10).

In Table 7.5, some sample rings with DFT parameters are given. We give an example to show how we get these figures. We first select a ring, for instance, let

Table 7.5 SMP parameter selection for SME.

Bits k	Ring $\mathbb{Z}_q$	DFT d	Root ω	Wordsize u	Words s
513	$(2^{57}-1)/7$	114	-2	9	57
518	$2^{73}-1$	73	2	14	37
704	$2^{64}+1$	128	2	11	64
1,185	$2^{79}-1$	158	-2	15	79
2,060	$(2^{103}+1)/3$	206	2	20	103
2,163	$2^{103}-1$	206	-2	21	103
3,456	$(2^{128}+1)$	256	2	27	128
4,260	$(2^{142}+1)/5$	284	2	30	142

us take $q = 2^{79} - 1$. This comes with the principal root of unity $\omega = -2$, the length $d = 158$ and $s = \lceil d/2 \rceil = 79$. Plugging these values into the Inequality (7.10) gives

$$138754.3b^4 + 277508.7b^3 + 138833.3b^2 < 2^{79} - 1$$

and then, by inspection, $b = 2^{15} \Rightarrow u = 15$ is found. Therefore, we may perform an exponentiation of maximum operand size equal to $k = s \cdot u = 79 \cdot 15 = 1185$ using SME with the specified parameters.

7.4.4 Parameter Selection for ECC over Prime Fields

An elliptic curve E over a prime field $GF(p)$, p odd prime, is determined by parameters $a, b \in GF(p)$ which satisfy $4a^3 + 27b^2 \neq 0$. The curve consists of the set of solutions or points $\mathbb{p} = (x, y)$ for $x, y \in GF(p)$ to the equation

$$y^2 \equiv x^3 + ax + b \mod p \tag{7.11}$$

together with an extra point $\mathbb{o}$ called the point at infinity. The set of points on E forms a group under the following addition rule: Let $(x_1, y_1) \in E(GF(p))$ and $(x_2, y_2) \in E(GF(p))$ be two points such that $x_1 \neq x_2$. Then, we have $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$, where

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \,, \\ y_3 &= \lambda(x_1 - x_3) - y_1 \,, \end{aligned}$$

where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$.

Observe that all computations are performed within the finite field $GF(p)$. Therefore, spectral modular algorithms of the previous sections can be used for field operations. In particular, SMP can be used for multiplications.

The security provided by ECC is guaranteed by the difficulty of the discrete logarithm problem in the elliptic curve group. The discrete logarithm problem is the problem of finding the least positive number, e, which satisfies the equation

$$\mathbb{q} = e \times \mathbb{p} = \underbrace{\mathbb{p} + \mathbb{p} + \cdots + \mathbb{p}}_{e \text{ times}} \,,$$

where $\mathbb{p}$ and $\mathbb{q}$ are points on the elliptic curve. Naturally, the basic computation (called *point multiplication*) in ECC is finding the eth (additive) power of an element $\mathbb{p}$ in the group. This involves additions, multiplications, and inversions of integers which are in the coordinates of the points. That is, it relies completely upon calculations in the underlying field, $GF(p)$.

Therefore, the elliptic point multiplication operation can be performed using the SMP consecutively. Once again, Inequality (7.10) helps us to find parameters for ECC use. In Table 7.6, some sample rings with DFT parameters are given.

Table 7.6 SMP Parameter selection for ECC use.

Bits k	Ring $\mathbb{Z}_q$	DFT d	Root ω	Wordsize u	Words s
513	$(2^{57}-1)/7$	114	-2	9	57
518	$2^{73}-1$	73	2	14	37
704	$2^{64}+1$	128	2	11	64
1,185	$2^{79}-1$	158	-2	15	79
2,060	$(2^{103}+1)/3$	206	2	20	103
2,163	$2^{103}-1$	206	-2	21	103
3,456	$(2^{128}+1)$	256	2	27	128
4,260	$(2^{142}+1)/5$	284	2	30	142

7.5 Spectral Extension Field Arithmetic

Spectral methods can be be applied to extension fields. Since binary and mid-size characteristic extensions are mostly of interest in practice, we breifly discuss these cases.

7.5.1 Binary Extension Fields

The most essential point of applying the spectral methods to binary field arithmetic is to find some suitable DFT domains having acceptable transform lengths for certain principal roots of unity. Unfortunately, if p is small, $\mathbb{Z}_p$ admits very short transform lengths (e.g., $\mathbb{Z}_2$ allows only a transform of length two). One way to overcome this problem is to use some polynomial rings over $\mathbb{Z}_p$ as the domain of DFT allows longer transform lengths because of their larger cardinality.

7.5.1.1 Suitable Polynomial Ring Spectrums

Spectral methods generally partition bigger problems into small pieces and then process the pieces in a parallel fashion. Notice that the computations in these pieces are carried in the ring, $R = \mathbb{Z}_2[\gamma]/(g(\gamma))$, hence for a proper $g(\gamma)$ selection, spectral methods benefit the most.

The most convenient choice of $g(\gamma)$ is a binomial. Moreover, if the principal root of unity, ω is chosen as a power of γ, the spectral coefficients are computed only using XORs and circular shifts. However, DFTs over polynomial rings having defining binomials suffer from the short transform lengths. For instance, $\gamma^n + 1$ has the linear factor $\Phi_1(\gamma) = \gamma + 1$ for all n, and by Theorem 1 of Pollard [2], only a transform length of two can be defined over these rings. Nevertheless, it is possible to overcome such restrictions using pseudotransforms (PT).

Pseudonumber transforms (PNT) are initially defined over subrings of Mersenne or Fermat rings. They support longer transform lengths and benefit the simplified arithmetic of the parent Mersenne or Fermat rings [10]. A similar approach can possibly be used for constructing pseudotransforms over polynomial rings. If $g(\gamma) = \gamma^n + 1$ is considered, a nice transform with a longer length can be grasped in a subring defined by a proper factor of $g(\gamma)$.

In general, factoring $\gamma^n + 1$ is not an easy problem which is closely related to the *cylotomic polynomials*. Since we are interested in binomials having fairly small degrees, even using a general purpose computer algebra system is satisfactory for our needs. Nevertheless, we present some pleasant arguments for the factors of cylotomic polynomials.

Definition 7.18. Let n be a positive integer, and let ω be primitive nth root of unity. The polynomial

$$\Phi_n(t) = \prod_{\gcd(n,k)=1} (t - \omega^n) \quad \text{for } 1 \leq k < n,$$

is called the nth cyclotomic polynomial .

The nth cyclotomic polynomial $\Phi_n(t)$ has degree $\varphi(n)$, where φ is the Euler's quotient function. These polynomials are irreducible over the rational numbers for every positive integer n but when they are considered over finite fields, this is no longer correct in general.

Since cyclotomic polynomials are minimal polynomials of the roots of unity, $g(\gamma) = \gamma^n \pm 1$ factor into cyclotomic polynomials. Consequently, the polynomial $\gamma^n - 1$ can be written as

$$\gamma^n - 1 = \prod_{d|n} \Phi_d(t).$$

Note that the above factorization is not necessarily prime over finite fields. For instance, $\gamma^5 - 1 = \phi_1(\gamma)\phi_4(\gamma)$ but $\phi_4(\gamma) = (t+1)^2$ over $GF(2)$. Letting p be an odd prime, some interesting examples over $GF(2)$ are as follows;

$$\begin{aligned} t^p + 1 &= \Phi_1(t)\Phi_p(t), \\ t^{2p} + 1 &= \Phi_1(t)\Phi_2(t)\Phi_p(t)\Phi_{2p}(t), \\ t^{4p} + 1 &= \Phi_1(t)\Phi_2(t)\Phi_p(t)\Phi_{2p}(t)\Phi_{4p}(t), \\ &\vdots \end{aligned}$$

One finds the first few remaining values of n as $t+1 = \Phi_1(t)$, $t^2+1 = \Phi_1(t)\Phi_2(t)$, $t^4+1 = \Phi_1(t)\Phi_2(t)\Phi_4(t)$, $t^8+1 = \Phi_1(t)\Phi_2(t)\Phi_4(t)\Phi_8(t)$ and $t^9+1 = \Phi_1(t)\Phi_3(t)\Phi_9(t)$.

Remark 7.7. In general, arithmetic in the factor rings is harder than the one in R, but being defined over a subring, PT calculations can be carried modulo $\gamma^n + 1$ for intermediate values and then the results are transformed to the factor ring by a final reduction. Such an approach simplifies the overall computation.

Example 7.7. Let us consider the DFT over $R = \mathbb{Z}_2[\gamma]/(\gamma^7 + 1)$ with the principal root of unity $\omega = \gamma$. Since $\gamma^7 + 1$ has the following factorization

$$\gamma^7 + 1 = \underbrace{(\gamma+1)}_{\Phi_1(\gamma)}\underbrace{(\gamma^3+\gamma^2+1)(\gamma^3+\gamma+1)}_{\Phi_7(\gamma)},$$

the ring R admits transform of lengths at the most two but if the ring $R' = \mathbb{Z}_2[\gamma]/(\Phi_7(\gamma))$, we get a 7-point DFT satisfying the convolution property over the ring R'. Besides that, one needs a $\Phi_7(\gamma)$ reduction while working in R' which is obviously harder than the arithmetic in R. However, since R' is a subring of R, all calculations can be carried over R with a final $\Phi_7(\gamma)$ whenever necessary.

In Table 7.7, we present the parameters for some suitable pseudotransform rings. One can find an appropriate $g(\gamma)$ by simply examining the transform length d in order to meet the marginal needs of a particular application.

Table 7.7 Suitable Polynomial Rings for an odd prime d.

Ring, $g(\gamma)$	ω	lenght
$(\gamma^d+1)/(\gamma+1), (\gamma^{2d}+1)/(\gamma^2+1)$	γ	d
$(\gamma^{d^2}+1)/(\gamma^d+1), (\gamma^{2d^2}+1)/(\gamma^{2d}+1)$	γ	d^2

Remark 7.8. While embedding the input to the pseudotransform domain, the size of the subring should be considered rather than the size of ring R. In fact, the most interesting pseudotransforms are the ones enlarging the lengths with minimal shrinkage in size. For further discussion we refer the reader to [11].

7.5.1.2 Suitable Finite Field Spectrums

We discuss the arithmetic simplifications when the factor rings are finite fields (i.e., defining polynomials are irreducible).

Note that binary extension fields can be seen as n-dimensional vector spaces over $GF(2)$: if $\{\alpha_1, \ldots, \alpha_n\}$ is taken as the basis set, each element of $GF(2^n)$ can be represented as a linear combination of the elements of this basis set. Among various bases, there are two special types having particular importance. The first one is the canonical polynomial basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{n-1}\}$, made up of powers of a defining (mostly primitive) element α of $GF(2^n)$. The second one is the normal basis of the form

$$N = \{\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}\} \tag{7.12}$$

and consists of a normal element $\alpha \in GF(q^n)$ and its conjugates with respect to $GF(2)$.

For every finite field there exists a normal basis, in fact, several such bases may exist for the same field. Those bases having the minimal complexity while multiplying field elements are the most important ones for computations (also called optimal normal bases (ONB)).

For our purposes type I ONBs have the utmost importance in which the element α is taken as the principal root of unity. Observe that this is the case where the basis (7.12) and the set of roots of unity (i.e., $\{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$) become set equivalent (not necessarily equal as ordered sets); hence, one can change the basis from normal to polynomial or vice versa by simply ordering the terms. Unfortunately, not all the finite fields have type I ONB; the following proposition gives a condition for their existence.

Proposition 7.7. *Suppose $n+1$ is a prime and q is primitive in $\mathbb{Z}_{n+1}$, where q is a prime or prime power. Then the n nonunit $(n+1)$th roots of unity are linearly independent and they form an ONB of $GF(q^n)$ over $GF(q)$.*

Proof. See Mullin et al. [12]

Using the above result, one can get that for $k = 4, 10, 12, 18, 28, 36, 52, 58, 60, \ldots$ the binary extension field $GF(2^k)$ has type I ONB.

In a normal basis representation, squaring a field element corresponds to a simple circular shift which seems well-suited for the realizations of public-key cryptosystems employing some form of repeated square and multiply methods, but in general, these representations mostly suffer from the complicated bases conversions and field multiplications. Eventually, type I ONB are optimal by giving the simplest conversion and multiplication realizations. Therefore, they initially favor a great interest in realizations of ECC but because of some security concerns, the use of elliptic curves over composite fields (type I ONB only exist in these extensions) is explicitly excluded from standards such as ANSI X9.63 [13].

Remark 7.9. We tend to choose a field having a type I ONB for transform domain. Observe that such a selection is implementation-related that does not change any ECC parameter, hence it never jeopardizes the security of the crypto- system.

7.5.1.3 Parameter Selection for ECC over Binary Fields

The size of the underlying structure (which also defines the key length) is a common security measure for public-key cryptosystems. After discarding the weak family of elliptic curves, standard documents [13] and [14] recommend some curves serving different needs of security levels. Referencing to the key sizes of these curves, in Table 7.8, we tabulate some suitable polynomial rings that admit nice DFT structures. Note that unlike SMM, when SMP is used for ECC, the word size $u \approx v/4$ as a result of successive SMP usage. In fact, a modification of SMP may give a much better u, (see the research project 1 of Section 7.8).

7.5.2 Midsize Characteristic Extension Fields

The ideas of the previous sections could be applied to the polynomial rings having midsize characteristics. In particular, extension fields $GF(q)$ with $q = p^k$, p an odd

Table 7.8 Standard Parameter Selection for SMP; † shows the domains having Type I ONB.

Bits k	PT ring $g(\gamma)$	DFT d	Root w	Wordsize u	Words s
171	$(\gamma^{37}+1)/(\gamma+1)$†	37	γ	9	19
210	$(\gamma^{41}+1)/(\gamma+1)$	41	γ	10	21
242	$(\gamma^{43}+1)/(\gamma+1)$	43	γ	11	22
288	$(\gamma^{47}+1)/(\gamma+1)$	47	γ	12	24
450	$(\gamma^{59}+1)/(\gamma+1)$†	59	γ	15	30
578	$(\gamma^{67}+1)/(\gamma+1)$†	67	γ	17	34

special prime, are good study cases. As we have seen in Section 7.5.1, when p is a small prime, DFTs suffer from short transform lengths. On the other hand, taking p such that $p \in [2^5, 2^{32}]$ gives great opportunities. Moreover, since the elements of the GF(q) could be represented by polynomials modulo p, one does not need to worry about the carries or coefficient overflows. In fact, this considerably simplifies SMP and any related computation. Let us start by giving the simplified SMP algorithm, and then we continue with further discussions.

7.5.2.1 Irreducible Binomials and Trinomials

As we mentioned in Section 7.5.1, the cryptosystems designed over extension fields give us the opportunity of choosing the parameter p and $f(t)$ freely. Certainly, we picked p as a Mersenne or Fermat prime or a large divisor of non-prime such numbers, and tend to choose $f(t)$ as a low hamming weight polynomial such as a binomial or a trinomial. Moreover, we insist on fixing the coefficients to powers of two, so that multiplications on the coefficients enjoy shifts instead of full multiplications.

We discussed the suitability of Mersenne and Fermat numbers earlier. Here, we start by giving the existence characterization of irreducible binomials. The next theorem is due to [15];

Spectral Modular Product
Assume that there exists a DFT map $DFT_d^{\omega} : \mathbb{Z}_p^d \rightarrow \mathscr{F}_p^d$, and $X(t), Y(t)$ and $F(t)$ are transform pairs of $x(t), y(t)$ and $f'(t)$ respectively, wherein, $x(t)$ and $y(t)$ are in the frame $\mathbb{Z}_p^s$ with $s = \lceil d/2 \rceil$, and $f'(t)$ is a multiple of the defining polynomial $f(t)$ in $\mathbb{Z}_p^{s+1}$ and $f'_0 = 1$.
Input: $X(t), Y(t)$ and $F(t)$; spectral polynomials
Output: $Z(t)$, spectral modular reduction of $x(t) \cdot y(t) \bmod f(t)$,
procedure SMP($X(t), Y(t)$)

1: $Z(t) := X(t) \odot Y(t)$
2: **for** $i = 0$ **to** $d-1$
3: $\quad z_0 := d^{-1} \cdot (Z_0 + Z_1 + \ldots + Z_d) \bmod p$
4: $\quad Z(t) := Z(t) - z_0 \cdot F(t) \bmod p$
5: $\quad Z(t) := Z(t) \odot \Gamma(t) \bmod p$
6: **end for**
7: **return** $Z(t)$

Theorem 7.8. *Let $l \geq 2$ be an integer and $a \in GF^*(q)$. Then the binary polynomial $t^l - a$ is irreducible in $GF(q)[t]$ if and only if the following conditions are satisfied: (i) each prime factor of l divides the order e of a in $GF^*(q)$ but not $(q-1)/e$; (ii) $q \equiv 1 \bmod 4$ if $l \equiv 0 \bmod 4$.*

Proof. See pages 124–125 of [15].

As a corollary we specify the existence of irreducible binomials over Mersenne fields.

Corollary 7.3. *Let $q = 2^r - 1$ be a Mersenne prime and $\omega = \pm 2$, the binomials, $t^r - \omega^i$ are irreducible in $GF(q)[t]$ if and only if r^2 does not divide $q-1$ and $gcd(r,i) = 1$ for $i = 1,2,\ldots,2r$.*

Proof. We simply check whether the conditions of Theorem 7.8 are satisfied or not. We start with condition (ii); r has to be odd in order that q be a prime. Hence, r does not divide 4 and condition (ii) is always satisfied.

For condition (i), the order of ω^i surely divides the order of ω which is $|\omega| = r$ or $2r$. Since the set of roots of unity forms a cyclic subgroup of order r or $2r$, those elements with power relatively prime to r or $2r$ have order equal to r or $2r$; others are proper divisors.

As an example; by using Corollary 7.3, the irreducible binomials in $GF(q)[t]$ for $q = 2^{13} - 1$ that interest us most are given simply the form $t^{13} - 2^i$ for $i \neq 13$ and $i \in \{1,2,\ldots,25\}$.

When trinomials are considered, it is hard to characterize the conditions compactly. Therefore, once again we refer the reader to [15] for further reading. In fact, since we are interested in relatively small degree polynomials and these polynomials are comparably dense in $GF(p)[t]$, searching methods are suitable for finding such polynomials. In order to give some samples, we tabulate such polynomials in Table 7.9.

7.5.2.2 SMP with Binomials or Trinomials

If special irreducible binomials or trinomials are used for SMP algorithms, a significant improvement is possible. To be more specific, in Step 3 of SMP method we

Table 7.9 Some irreducible trinomials in $GF(q)[t]$ for $q = 2^{13} - 1$.

$t^{13}+t+2$	$t^{13}+t+2^{10}$	$t^{13}+t+2^{19}$
$t^{13}+t+2^2$	$t^{13}+t+2^{11}$	$t^{13}+t+2^{20}$
$t^{13}+t+2^3$	$t^{13}+t+2^{12}$	$t^{13}+t+2^{21}$
$t^{13}+t+2^4$	$t^{13}+t+2^{13}$	$t^{13}+t+2^{22}$
$t^{13}+t+2^5$	$t^{13}+t+2^{14}$	$t^{13}+t+2^{23}$
$t^{13}+t+2^6$	$t^{13}+t+2^{15}$	$t^{13}+t+2^{24}$
$t^{13}+t+2^7$	$t^{13}+t+2^{16}$	$t^{13}+t+2^{25}$
$t^{13}+t+2^8$	$t^{13}+t+2^{17}$	$t^{13}+t+2^{26}$
$t^{13}+t+2^9$	$t^{13}+t+2^{18}$	

subtract the z_0 multiple of $F(t)$ from the partial sum. If $f(t)$ is a random irreducible polynomial, this multiplication corresponds to a $v \times v$ multiplication but with the special trinomials or binomials this multiplication is performed by simple shifts.

Let $f(t) = t^m + \omega^{-s_0}$ with an integer s_0 be an irreducible binomial, $f'(t)$ simply equals $f'(t) = 1 + \omega^{s_0} t^m$. If the transform pair of $f'(t)$ is computed, one gets

$$F(t) = 1 + \omega^{s_0} + (1 + \omega^{s_0+m})t + \cdots + (1 + \omega^{s_0+m(d-1)})t^{d-1}.$$

Hence it is easily seen that the Step 3 of SMP follows

$$\begin{aligned} z_0 \cdot F_i &= z_0(1 + \omega^{s_0+mi}) \\ &= z_0 + z_0\omega^{s_0+mi} \end{aligned}$$

for $i = 0, 1, \ldots, d-1$. Observe that all the $z_0\omega^{s_0+mi}$ are computed by simple shifts because $\omega = 2$. Similarly, if $f(t)$ is a trinomial, another shift-add has to be performed.

7.5.3 Parameter Selection for ECC over Extension Fields

Spectral multiplication can be extremely efficient for extension fields having medium characteristics. By "medium" we mean the typical wordsize of today's architectures. For instance, if the field $GF(p^k)$ is considered we assume $2^7 < p < 2^{32}$.

In the literature, the security of an ECC employment is given according to the length of the key sizes. These key sizes are determined according to the complexities of the best-known algorithms known for solving the discrete logarithm problem in elliptic curve groups over the fields $GF(p^k)$. In Table 7.10, we tabulated the parameter selection of some nice Mersenne and Fermat fields that target some popular key sizes.

As mentioned in Section 7.4.2, psuedotransforms are also very convenient for employing spectral algorithms. In this context, if the prime p is chosen to be a

Table 7.10 Parameter Selection for ECC over $GF(p^k)$.

Bits	$GF(p^k)$	DFT	Root	Wordsize	Words
$s \cdot u$	p	d	ω	v	$s = k$
153	$2^{17}-1$	17	2	17	9
169	$2^{13}-1$	26	-2	13	13
190	$2^{19}-1$	19	2	19	10
256	$2^{16}+1$	32	2	16	16
289	$2^{17}-1$	34	-2	17	17
361	$2^{19}-1$	38	-2	19	19
496	$2^{31}-1$	31	2	31	16
512	$2^{16}+1$	64	$\sqrt{2}$	16	32
961	$2^{31}-1$	62	-2	31	31

Table 7.11 Parameter Selection for ECC over $GF(p^k)$ using PNTs.

Bits	$GF(p^k)$	DFT	Root	Wordsize	Words
$s \cdot u$	k	d	ω	u	s
150	$(2^{20}+1)/17$	20	4	15	10
170	$(2^{19}+1)/3$	19	4	17	10
255	$(2^{17}+1)/3$	34	-2	15	17
323	$(2^{19}+1)/3$	38	-2	17	19
352	$(2^{32}+1)/641$	32	4	22	16
464	$(2^{31}+1)/3$	31	4	29	16
483	$(2^{23}+1)/3$	46	2	21	23
644	$(2^{28}+1)/17$	56	2	14	23
899	$(2^{31}+1)/3$	62	2	29	31

divisor of the psuedo-Mersenne or Fermat number, one gets the parameters in Table 7.11 for efficient implementations. Although arithmetic modulo p might be difficult, the actual computation is carried in the chosen Mersenne or Fermat ring with a final modulo p reduction.

7.6 Notes

In this chapter new techniques of performing modular multiplication and exponentiation are proposed. Especially, modular exponentiation is one of the most important arithmetic operations for methods of modern cryptography, such as the RSA and Diffie-Hellman algorithms. The proposed methods use the Discrete Fourier Transform over finite rings, and relies on new techniques to perform the modular reduction operation.

The wonders of the convolution property has been known over decades. Obtaining modular arithmetic algorithms fully working in the spectrum would benefit the convolution property to the maximum extent. For carrying modular arithmetic, one need obviously has to deal with the concept of modular reduction. In [10] and later in a more compact text [16], after defining the spectral reduction and related concepts, a spectral reduction algorithm is introduced using the linearity and shifting property of DFT. Spectral modular multiplication (SMM) and spectral modular exponentiation (SME) come quite naturally once a reduction is defined.

When it comes to the practicability of the proposed methods, there were many directions to go because of the richness of the spectral theory. A first experiment could possibly work in a complex spectrum but, because of massive computations in the spectrum, the round-off errors could be hard to control (but still an analysis is needed). Therefore, a smart move is to employ the finite ring spectrums for not admitting the round-off errors in the computations. Additionally, from a computational point of view, calculations in some special rings such as Fermat and Mersenne can

exploit the special arithmetic. In fact, there are excellent references [17], [18], [19] and [20] demonstrating efficient arithmetic in these rings. Moreover, one can check [4] for arithmetic in pseudotransform arithmetic.

The ideas utilized for modular operations of large integers are extendable to polynomial rings. The extension to the rings having mid-size characteristic is the easiest by simply using the underlying ring as the domain of the DFT. In fact, because of this simplicity, one does not need to worry about the carries or coefficient overflows and can have a very convenient method for the ring arithmetic. The method is independently proposed in [21] and [22]. Later a coprocessor [23] based on the method is introduced.

On the other hand, because of very short transform lengths (e.g., the ring $\mathbb{Z}_2$ allows only a transform two length of) using the spectral methods for binary or small characteristic ring extensions is a little cumbersome. One way to overcome this problem is to use some polynomial rings over $\mathbb{Z}_p$ as the domain of DFT allowing longer transform lengths [11] because of their larger cardinality.

Because of working in the spectrum, there exists a vast amount of parallelism potential in computations. Therefore, these methods have the chance of yielding efficient and highly parallel architectures especially for hardware implementations. Although we do not discuss implementation aspects in this text, the reader could find architectures and unit-gate analysis of the described methods in [10], [16], [11], [21] and [23].

7.7 Exercises

1. What is the maximum DFT lenght that can be defined over the ring $\mathbb{Z}_{2^{20}+1}$, $\mathbb{Z}_{2^{31}-1}$ and $\mathbb{Z}_{2^{79}-1}$? What will be this maximum if the principle root of unity is an integer power of two?
2. What are the best pseudotransform rings for $\mathbb{Z}_{2^{20}+1}$, $\mathbb{Z}_{2^{25}-1}$ and $\mathbb{Z}_{2^{39}+1}$ maximizing the DFT length? What will be these maximum lengths if the principle root of unity is an integer power of two?
3. What is the maximum modulus size that can be used for SME over the ring $\mathbb{Z}_{2^{20}+1}$? What will be this maximum if the principle root of unity is an integer power of two?
4. Assume that we want to use SME for an RSA system having a 1100 bits modulus. What is the smallest Mersenne and Fermat ring for the DFT such that SME works without overflows, if the principle root of unity is chosen as integer power of two.
5. Calculate $c = m^e \pmod{n}$ for $m = 2718$, $e = 53$, and $n = 3141$ using SME over the ring $\mathbb{Z}_{2^{19}-1}$.
6. What is the maximum DFT length (and relative the principal root of unity) that can be defined over the ring $R = \mathbb{Z}_2[\gamma]/(g(\gamma))$ where $g(\gamma) = (\gamma^{29}+1)$, $g(\gamma) = (\gamma^{29}+1)/(\gamma+1)$, $g(\gamma) = (\gamma^{49}+1)$ and $g(\gamma) = (\gamma^{49}+1)/(\gamma^7+1)$? What will be this maximum if the principal root of unity is power of γ?

7. What is the biggest binary field which its arithmetic can be carried using DFT over the ring $R = \mathbb{Z}_2[\gamma]/(g(\gamma))$, where $g(\gamma) = (\gamma^{19}+1)/(\gamma+1)$? What will be this field if the principal root of unity is a power of γ?
8. In characteristic three rings, DFT also suffers from short lengths. However, spectral methods can be applied similar to binary extensions. What is the maximum DFT length (and relative the principal root of unity) that can be defined over the ring $R = \mathbb{Z}_3[\gamma]/(g(\gamma))$, where $g(\gamma) = (\gamma^{23}+1)$, $g(\gamma) = (\gamma^{23}+1)/(\gamma+1)$, $g(\gamma) = (\gamma^{31}+1)$ and $g(\gamma) = (\gamma^{49}-1)/(\gamma^7-1)$? What will be this maximum if the principal root of unity is power of γ?
9. Assume that we want to employ a DFT for arithmetic in $GF(p^k)$ What is the maximum DFT length that can be defined over the ring $\mathbb{Z}_{2^{13}+1}$, $\mathbb{Z}_{2^{13}-1}$ and $\mathbb{Z}_{2^{79}-1}$? What will be this maximum if the principal root of unity is an integer power of two?
10. Let $m(t) = 6+3t+2t^2+5t^3 \in GF(p^k)$ for $p = 2^7-1$ and $k = 7$. If $f(t) = t^7-2$ is the defining polynomial for $GF(p^k)$ then calculate $c(t) = (m(t))^e \in GF(p^k)$ for $e = 53$.

7.8 Projects

1. If the SMP (i.e., Algorithm 7.5.2.1) is considered, notice that our bound analysis depends heavily on $\beta \cdot \underline{N}(t)$ multiplication of Step 7. In fact, it is possible to replace this multiplication by a multioperand addition at a cost of some precomputations and extra memory.
   To be more specific, let $b = 2^u$ and $n_i(t)$ be the polynomial representation of an integer multiple of n such that the zeroth coefficient of $n_i(t)$ satisfies $(n_i)_0 = 2^{i-1}$ for $i = 1,2,\ldots,u$ (note that $\underline{n}(t) = n_1(t)$). We can now write $\beta \cdot \underline{N}(t)$ as

$$\beta \cdot \underline{N}(t) = \sum_{i=1}^{u} \beta_i \cdot N_i(t) \ , \tag{7.13}$$

   where β_i is a binary digit of β and $N_i(t) = \mathrm{DFT}_d^{\omega}(n_i(t))$ for $i = 1,2,\ldots,u$. Note that $\beta < b$ and $\beta_i = 0$ for $i \geq u$.
   Plugging the Equation (7.13) into the Algorithm 7.5.2.1 gives a modified spectral modular product (MSMP) algorithm. Observe the benefit of this approach since this replacement gives a reasonable amount of radius shrinkage. Calculate the new bound with respect to this modified algorithm.
2. In many situations it is desirable to break a congruence mod n into a system of small congruences modulo factors of n. Once necessary computations are performed in the small factor rings, using CRT , the resultant system of congruences is replaced by a single congruence under certain conditions.
   When spectral algorithms are considered, CRT can be used in two different ways. The first one is for degree that can be adopted from Quisquater and Couvreur [24] where the second one is for radius that is based on the ideas proposed for integer

multiplication by J. M. Pollard [25], and independently by A. Schönhage and V. Strassen [1].
Examine these two utilizations and present algorithms for both methods. Give a boundary anaylsis for both of the algorithms. Give a parameter selection table for popular RSA sizes using SMP and MSMP.

3. Note that Mersenne arithmetic corresponds to one's complement arithmetic whereas Fermat arithmetic may be implemented in different fashions. Research on efficient Fermat ring arithmetic ([20] and [17] could be two decent starting articles), then design parallel and digit serial Mersenne and Fermat multipliers for $\mathbb{Z}_{2^{13}-1}$ and $\mathbb{Z}_{2^{24}+1}$ respectively. Plot the relation between the area and digit size.
4. Consider the Fermat ring $\mathbb{Z}_{2^{24}+1}$. If $\omega = 8$ is taken, one gets a DFT having length 16. Calculate the parameters u and b in order to determine the maximum supported RSA length. Design a hardware architecture for SMP over the Fermat ring $\mathbb{Z}_{2^{24}+1}$.
5. Consider the Mersenne ring $\mathbb{Z}_{2^{13}-1}$. If $\omega = -2$ is taken, one gets a DFT having length 26. Design a an SMP architecture over the ring $\mathbb{Z}_{2^{13}-1}$ performing $GF(p^k)$ arithmetic for $p = 2^{13} - 1$, $k = 13$ and $f(t) = t^{13} - 2$ is the defining polynomial.

References

1. A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7: 281–292, 1971.
2. J. M. Pollard. Implementation of number theoretic transform. *Electronics Letters*, 12(15): 378–379, July 1976.
3. R. E. Blahut. *Fast Algorithms for Digital Signal Processing*, Addison-Wesley publishing Company, 1985.
4. H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*, Springer, Berlin, Germany, 1982.
5. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2): 120–126, February 1978.
6. T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings – Computers and Digital Techniques*, 149(2): 46–52, March 2002.
7. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170): 519–521, April 1985.
8. Ç. K. Koç. High-Speed RSA Implementation. Tech. Rep. TR 201, RSA Laboratories, 73 pp. November 1994.
9. N. Koblitz. *A Course in Number Theory and Cryptography*, Springer, Berlin, Germany, Second edition, 1994.
10. G. Saldamlı. *Spectral Modular Arithmetic*, Ph.D. thesis, Department of Electrical and Computer Engineering, Oregon State University, May 2005.

11. G. Saldamlı and Ç. K. Koç. Spectral modular arithmetic for binary extension fields. *preprint*, 2006.
12. S. A. Vanstone, R. C. Mullin, I. M. Onyszchuk and R. M. Wilson. Optimal normal bases in GF(p^k). *Discrete Applied Mathematics*, 22: 149–161, 1989.
13. ANSI X9.62-2001. Public-key cryptography for the financial services industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography. 2001, Draft Version.
14. IEEE. *P1363: Standard specifications for public-key cryptography*. November 12, 1999, Draft Version 13.
15. R. Lidl and H. Niederreiter. *Finite Fields*, Encyclopedia of Mathematics and its Applications, Volume 20. Addison-Wesley publishing Company, 1983.
16. G. Saldamlı and Ç. K. Koç. Spectral modular arithmetic. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic 2007 (ARITH'07)*, 2007, pp. 123–132.
17. J.-L. Beuchat. A family of modulo (2^n+1) multipliers, *Tech. Rep. 5316*, Institut National de Recherche en Informatique et en Automatique (INRA), September 2004.
18. Z. Wang, G. A. Jullien, and W. C. Miller. An efficient tree architecture for modulo 2^n+1 multiplication. *J. VLSI Signal Processing Systems*, 14(3): 241–248, December 1996.
19. R. Zimmermann. "Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication," in *Proceedings of the 14th IEEE Symposium on Computer Architecture*, 1999, pp. 158–167.
20. L. M. Leibowitz. A simplified binary arithmetic for the Fermat number transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24: 356–359, 1976.
21. G. Saldamlı and Ç. K. Koç. Spectral modular arithmetic for polynomial rings. *preprint*, 2006.
22. S. Baktir and B. Sunar. Finite field polynomial multiplication in the frequency domain with application to Elliptic Curve Cryptography. In *Proceedings of Computer and Information Sciences ISCIS 2006)*, pp. 991–1001, 2006.
23. S. Baktir, S. Kumar, C. Paar, and B. Sunar. A state-of-the-art elliptic curve cryptographic processor operating in the frequency domain. *Mobile Networks and Applications (MONET)*, 12(4): 259–270, September 2007.
24. J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21): 905–907, Oct. 1982.
25. J. M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25: 365–374, 1971.

Chapter 8
Elliptic and Hyperelliptic Curve Cryptography

Nigel Boston and Matthew Darnall

8.1 Introduction

Suppose two parties, Alice (A) and Bob (B), want to send messages between themselves without an eavesdropper Eve (E) reading the messages. Private-key (symmetric) cryptography relies on establishing a known secret between A and B before they can communicate. The term symmetric describes the fact that the information known to A and B is the same, namely the private key. We have seen an example of a private-key system, advanced eneryption standard (AES), in chapter 1. What if, as often happens in practice, it is infeasible for A and B to have a prearranged secret? In the development of cryptography it became apparent that a mechanism for A and B to agree upon a private key over an insecure channel would be important.

The area of cryptography devoted to the ways Alice and Bob can share information without a prearranged secret is called public-key (or asymmetric) cryptography. The term public key refers to the fact that in all current systems, some public piece of information is needed for the encryption to occur. Examples of this public information are the modulus in the famous RSA algorithm [46] or the group generator raised to a power for Diffie – Hellman, described later. The term asymmetric means that the private information known to A and B is different, i.e., A and B each start with information the other does not know. Public-key cryptography was introduced to the world at large in the seminal paper [10] of Whitfield Diffie and Martin Hellman in 1978, although shortly before then these ideas were known to the researchers at the British intelligence agency GCHQ*. Many other methods of public-key cryptography have since been introduced and current research is still searching for better protocols for the exchange of private keys.

University of Wisconsin,
e-mail: boston,darnall@math.wisc.edu

*(UK) Government Communications Headquarters

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_8,

© Springer Science+Business Media, LLC 2009

8.2 Diffie – Hellman Key Exchange

Though it was the first method of public-key cryptography known, the Diffie – Hellman key exchange protocol is still used widely and makes up the basis for both elliptic and hyperelliptic curve cryptography. Let G be a finite cyclic group of order n with generator g. The discrete logarithm problem (DLP) for the group G is to determine a given g^a, where a is a positive integer less than n. The simple idea observed by these researchers is that, if there is a group where performing the group operation is computationally easy, but solving the DLP is hard, then a secret between two parties can be shared. A and B come up with private keys k_A and k_B respectively – these are positive integers less than n. A publishes g^{k_A} and B publishes g^{k_B}. The shared secret between A and B is $g^{k_A k_B}$, which A computes as $(g^{k_B})^{k_A}$ and B computes as $(g^{k_A})^{k_B}$. Since Eve apparently cannot get k_A or k_B without solving a DLP, obtaining the shared secret is hard. It is widely believed that solving the DLP is equivalent to determining $g^{k_A k_B}$ given g, g^{k_A}, g^{k_B}, although this is not known.

The groups originally considered for Diffie–Hellman Key Exchange were large cyclic subgroups of multiplicative groups of finite fields. As seen in Chapter 7, the multiplication in finite fields can be efficiently computed. Unfortunately, the DLP in this case can be solved in time subexponential in the size of the group using an index calculus attack. A result of Victor Shoup [52] says that for an arbitrary group of order n, computing a discrete log will take $\sqrt{p}$ group operations, where p is the largest prime divisor of n. This result assumes that no structure of the group can be taken advantage of, so groups attackable only in exponential time should exist. Currently, the most suitable groups that provide quick encryption and for which only exponential time attacks are known, come from elliptic and hyperelliptic curves.

8.3 Introduction to Elliptic and Hyperelliptic Curves

Elliptic curves have been studied by mathematicians for centuries. Neal Koblitz and Victor Miller independently discovered that their rich structure makes them useful for a wide range of cryptographic applications [26], [38]. This structure can also lead to several attacks, so care must be taken by any would-be cryptographer.

Let $\mathbf{k}$ be a field. An *elliptic curve* E, over $\mathbf{k}$, is a non-singular projective curve of genus 1. For an arbitrary field $\mathbf{k}$, E can be thought of as the set of points $(X,Y) \in \mathbf{k}^2$ that satisfy an equation of the form:

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

where the coeffients a_i are in $\mathbf{k}$, together with a "point at infinity," P_∞. We must also assume that the curve defined by the equation is non-singular, which is equivalent to having the partial derivatives of the equation never vanish simultaneously. If the characteristic of $\mathbf{k}$ is not 2 or 3, we can perform a change of variables to get an equation to the form

$$Y^2 = X^3 + aX + b$$

Here the non-singular condition just says that the cubic $X^3 + aX + b$ must have distinct roots.

A hyperelliptic curve is a special type of non-singular, projective curve. For our purposes, a *hyperelliptic curve*, of genus $g \geq 1$ over $\mathbf{k}$ is the set of points $(X,Y) \in \mathbf{k}^2$ that satisfy

$$y^2 + h(X)Y = f(X)$$

where h and f are polynomials in $\mathbf{k}[X]$ with $deg(f) = 2g+1$, $deg(h) \leq g$, together with a point "at infinity", P_∞. An elliptic curve is just a hyperelliptic curve of genus 1.

To understand why there is a point at infinity, notice that the definition of elliptic or hyperelliptic curves includes the word 'projective'. We consider the curves as living in projective space, say with coordinates $(X : Y : Z)$. The point at infinity is the unique point where the projective curve defined by homogenizing our equation intersects the line $Z = 0$. If the reader has no background in projective geometry, simply think of the point P_∞ as a point infinitely far up the Y-axis that 'compactifies' the curve.

8.4 The Jacobian of a Curve

A priori, a hyperelliptic curve over a field $\mathbf{k}$, is a set of points in $\mathbf{k}^2$ with a special point at infinity. A beautiful fact noticed a long time ago by algebraic geometers is that a group can be attached to each curve. These groups are "made up" of collections of points on the curve and the group law can be performed using only operations in the field $\mathbf{k}$. For an elliptic curve, the group consists of the points on the elliptic curve, and the group law can be viewed geometrically. For hyperelliptic curves, the group consists of g-tuples of points on the curve. For both kinds of curves, the DLP is in general very hard to solve.

The group associated to a hyperelliptic curve, C, is a quotient of the larger group called the *degree zero divisor group* of C, denoted $Div^0(C)$. This group is made up of elements, D, called divisors, of the form:

$$D = \sum_{P \in C} n_P P$$

where:

1. The formal sum is over points $P = (x_P, y_P)$ on C with coordinates x_P, y_P in $\bar{\mathbf{k}}$, an algebraic closure of $\mathbf{k}$. Here P_∞ is included as a point on C.
2. n_P is an integer for each P, with all but finitely many $n_P = 0$.
3. If there is an element σ of the Galois group of $\bar{\mathbf{k}}$ over $\mathbf{k}$ such that $\sigma P := (\sigma(x_P), \sigma(y_P)) = (x_Q, y_Q) = Q$, then $n_Q = n_P$.
4. $deg(D) = \sum_P n_P = 0$.

For readers unfamiliar with Galois groups, an element σ, of the Galois group of $\bar{\mathbf{k}}$ over $\mathbf{k}$ is an automorphism of $\bar{\mathbf{k}}$ that fixes $\mathbf{k}$. Thus, for any $a,b \in \bar{\mathbf{k}}$, we have $\sigma(a+b) = \sigma(a) + \sigma(b)$ and $\sigma(ab) = \sigma(a)\sigma(b)$. We also have that $\sigma k = k$ for any $k \in \mathbf{k}$.

We add two divisors by adding the coefficients corresponding to each point:

$$\sum_P m_P P + \sum_P n_P P = \sum_P (m_P + n_P) P$$

Notice that the new divisor still satisfies the conditions above.

8.4.1 The Principal Subgroup and $Jac(C)$

Let $F(X,Y) = Y^2 + h(X)Y - f(X)$ be the polynomial defining the curve C. Let $p(X,Y) \in \mathbf{k}[X,Y]$ be a polynomial in X and Y with coefficients in $\mathbf{k}$ that is not divisible by F. We shall get a divisor, $div(p) \in Div^0(C)$ from this polynomial. For every point $P = (x_P, y_P)$, we define $ord_p(P)$ to be the order at which p vanishes at P. This order has a rigorous definition that is beyond the scope of the book. Loosely speaking, at each point P, we can define something analogous to the Taylor expansion of the function p on C. The order, $ord_P(p)$, at which p vanishes at P is then the smallest exponent in the 'Taylor expansion' with a nonzero coefficient. Thus, $ord_P(p) = 0$ if and only if $p(x_P, y_P) \neq 0$. The order of a function can also be computed at P_∞, and in fact $ord_{P_\infty}(p)$ is the unique integer such that $\sum_{P \in C} ord_P(p) = 0$.

Definition 8.1. The divisor $div(p)$ associated to a polynomial $p(X,Y)$, p not divisible by F, is:

$$\sum_P ord_p(P) P$$

This divisor satisfies the conditions above to be an element of $Div^0(C)$.

We call a divisor principal if it can be written as $div(p) - div(q)$ for two polynomials p,q as above. The set of principal divisors forms a subgroup of $Div^0(C)$, denoted $Prin(C)$.

Definition 8.2. The quotient group $Div^0(C)/Prin(C)$, is called the *Jacobian* of C and is denoted $Jac(C)$.

If $\mathbf{k}$ is a finite field, as it will be for our purposes, then $Jac(C)$ is finite. It is this group that we use for our Diffie – Hellman key exchange.

8.5 Computing on $Jac(C)$

By the celebrated theorem of Riemann-Roch, which is beyond the scope of this book, we know a lot about the structure of $Jac(C)$. Namely, if C has genus g, every element of $Div^0(C)$ is equivalent in $Jac(C)$ to exaclty one divisor of the form:

$$\sum_{i=1}^{m} P_i - mP_\infty$$

where $P_i = (x_i, y_i)$, $m \leq g$, and $P_i \neq (x_j, -y_j - h(x_j)) = -P_j$ for $i \neq j$. When $m = 0$, we get the identity element of the group, the divisor with all coefficients equal to zero. Divisors of the above form are called *reduced* divisors. This gives us a method for representing points in $Jac(C)$ as unordered g-tuples of points on the curve over $\bar{\mathbf{k}}$. The condition that $n_P = n_Q$ for Galois conjugates P and Q ensures that every point occurring in a reduced divisor of $Jac(C)$ has coordinates in at most a degree g extension of $\mathbf{k}$. Thus, when $g = 1$ and we have an elliptic curve E, then the group $Jac(E)$ consists of the points on the curve E over $\mathbf{k}$.

It is easy to see that, if we add two reduced divisors D_1 and D_2 in the fashion described above, we are not guaranteed that the new divisor $D_1 + D_2$ will be reduced. We need a method for finding the unique reduced divisor corresponding to $D_1 + D_2$. To do this, we use a different representation for a divisor than the one given above. As before, let $Y^2 + h(X)Y = f(X)$ be the defining equation for C.

Definition 8.3. The Mumford representation of a reduced divisor $\sum_{i=1}^{m} P_i - mP_\infty$, $P_i = (x_i, y_i)$ is the unique pair of polynomials $[u(x), v(x)]$ in $\mathbf{k}[x]$ that satisfy the following:

1. $u(x) = \prod_{i=1}^{m}(x - x_i)$.
2. $deg(v) < deg(u) = m$.
3. $v(x_i) = y_i$.
4. $u(x)$ divides $v(x)^2 + h(x)v(x) - f(x)$.

The fact that the Mumford representation is unique follows from the fact that the m coefficients defining v can be determined uniquely by conditions 3 and 4. When $m = 0$, we have the identity element, and the Mumford representation is $[1, 0]$. Also every pair of polynomials $[u(x), v(x)]$ in $\mathbf{k}[x]$ satisfying

1. $deg(u) \leq g$ and $deg(v) < deg(u)$,
2. $u(x)$ divides $v(x)^2 + h(x)v(x) - f(x)$,
3. $u(x)$ is monic,

corresponds to a unique reduced divisor D. If $u(x) = \prod_{i=1}^{m}(x - x_i)$, then the points that make up D are $P_i = (x_i, v(x_i))$. Condition 2 will guarantee that that these points are on the curve C.

The benefit of using the Mumford represention for reduced divisors is the following algorithm for computing the sum of two reduced divisors. The algorithm is originally due to Cantor [5] for $h(X) = 0$, and in its most general form it is due to Koblitz [27]. The proof of the correctness of the algorithm, which we do not cover, can be found in [36].

Algorithm

Input: Two reduced divisors $D_1 = [u_1, v_1]$ and $D_2 = [u_2, v_2]$ given in Mumford representation.

Output: A reduced divisor $D = [u, v]$ in Mumford representation that satisfies $D = D_1 + D_2$ in $Jac(C)$.

1. $d_1 \leftarrow gcd(u_1, u_2) = e_1u_1 + e_2u_2$
2. $d \leftarrow gcd(d_1, v_1 + v_2 + h) = c_1d_1 + c_2(v_1 + v_2 + h)$
3. $s_1 \leftarrow c_1e_1$, $s_2 \leftarrow c_1e_2$, $s_3 \leftarrow c_2$
4. $u \leftarrow (u_1u_2)/d^2$
5. $v \leftarrow (s_1u_1v_2 + s_2u_2v_1 + s_3(v_1v_2 + f))/d \bmod u$
6. **DO**
7. $u' \leftarrow (f - vf - v^2)/u$
8. $v' \leftarrow (-h - v) \bmod u'$
9. $u \leftarrow u'$ and $v \leftarrow v'$
10. **WHILE** $deg(u) \leq g$
11. make u monic by dividing by the leading coefficient
12. return $[u, v]$

A key fact to notice is that all the operations involved in adding the two reduced divisors can be reduced to multiplication, division and addition of polynomials in $\mathbf{k}[x]$. Thus, using the techniques for finite field arithmetic given in the previous chapters, we can perform the group operations on $Jac(C)$ quickly. Cantor's algorithm given above is completely general; it works for any hyperelliptic curve over any field. In a practical implementation, properties of the curve and field are used to speed up the algorithm.

8.6 Group Law for Elliptic Curves

Computations in the Jacobian of an arbitrary hyperelliptic curve can be complicated. In this section, we give a simple geometric interpretation of the group law for elliptic curves. We give explicit algorithms for adding two points on an elliptic curve. The following chapter covers the various speedups and optimizations in more detail. The reader interested in implementation specifics should consult that chapter.

Let E be an elliptic curve over a field $\mathbf{k}$. We assume that the field $\mathbf{k}$ has characteristic not equal to 2 or 3, so that we can make a change of coordinates to make the defining equation for E of the form

$$Y^2 = X^3 + aX + b$$

with $a, b \in \mathbf{k}$. Elliptic curves over characteristic two fields are important for cryptographic uses, but for the geometric description of the group law it is easier to assume $char(\mathbf{k}) > 3$.

The map that sends a point P to the reduced divisor $P - P_\infty$ is a bijection between the points on E and $Jac(E)$, the Jacobian of E. We use this bijection and the definition of $Jac(E)$ to give a geometric meaning to the sum of two points. Recall that two divisors D_1 and D_2 are equivalent if $D_1 = D_2 + div(f) - div(g)$, where f and g are polynomials in $\mathbf{k}[X, Y]$ not divisible by $Y^2 - X^3 - aX - b$.

For two constants m, $c \in \mathbf{k}$, consider the line $Y = mX + c$ in the same plane as the elliptic curve E. By Bezout's theorem, we know that the line intersects E in

exactly three points, if we count the points with appropriate multiplicity. So the function $f(X,Y) = Y - mX - c$ has three (not necessarily unique) points on E where it vanishes. We can then write the divisor of f as

$$div(f) = P_1 + P_2 + P_3 - 3P_\infty$$

where P_1, P_2 and P_3 are the three (not necessarily unique) points where the line intersects E. The three points can be non-unique in only certain examples, such as when the line lies tangential to the curve E. In this case, the line intersects the curve E in only two actual points, though the tangential point has multiplicity 2. In analogy with the Taylor series, this is because at the tangential point, f and $Y^2 - X^3 - aX - b$ not only have the same value, they also have the same first derivative.

Since $P_1 + P_2 + P_3 - 3P_\infty = (P_1 - P_\infty) + (P_2 - P_\infty) + (P_3 - P_\infty)$ is the divisor of a function, it represents the identity in the group $Jac(E)$. Using the given bijection with E, we see that

Lemma 8.1. *The sum of two points P_1 and P_2 on E is equal to $-P_3$, where P_3 is the unique other point on E that intersects the line through P_1 and P_2.*

The question remains of what $-P_3$ means, i.e., what is the inverse of a point on E? We first notice that our bijection with $Jac(E)$ sends P_∞ to the divisor $P_\infty - P_\infty$, which is the identity of $Jac(E)$. Thus, P_∞ is the identity of E. Now, let $P = (x_P, y_P)$ be a point on E. Consider the function $X - x_P$. This function intersects the curve E at the points $P = (x_P, y_P)$, $Q = (x_P, -y_P)$, and P_∞. The first two interesect points are easy to see, the last one follows from looking at the projectivization of the curve and the line. Thus, in $Jac(E)$, the divisor $P + Q - 2P_\infty$ equals the identity, so $Q = -P$ on E.

The work above gives us the following algorithm for adding two points, P_1 and P_2. We simply take the unique line through P_1 and P_2, find the unique other point, (x,y), that is on the elliptic curve and the line, and return $P = (x, -y)$. Remember that if $P_1 = P_2$, the line through the point should have the same slope as the defining equation for E.

Algorithm

Input: Two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on the curve E defined by $Y^2 = X^3 + aX + b$.

Output: The point $P = (x,y) = P_1 + P_2$.

1. IF P_1 or $P_2 = P_\infty$ THEN $P \leftarrow P_2$ or P_1.
2. IF P_1 = P_2
3. IF $y_1 = 0$ THEN $P \leftarrow P_\infty$
4. $\lambda \leftarrow (3x_1^2 + a)/2y_1$
5. $x \leftarrow \lambda^2 - 2x_1$
6. $y \leftarrow \lambda(x_1 - x_3) - y_1$
7. IF $P_1 \neq P_2$
8. $\lambda \leftarrow (y_1 - y_2)/(x_2 - x_1)$
9. $x \leftarrow \lambda^2 - x_1 - x_2$
10. $y \leftarrow \lambda(x_1 - x_3) - y_1$

8.7 Techniques for Computations in Hyperelliptic Curves

Optimization of hyperelliptic curve arithmetic is a current area of research with several papers appearing each year in the top cryptography conferences. This section gives a brief survey of the techniques and ideas behind them. The reader interested in implementing one of the methods should consult the references given. As the genus gets larger, the computational costs rise significantly. This computational cost, as well as the existence of subexponential time index calculus attacks on high genus hyperelliptic curves, makes low genus curves the most practical for cryptography.

8.7.1 Explicit Formulae

As we showed with elliptic curves, the group law can be implemented using only additions, multiplications and inversions in the base field, **k**. To improve the runtime of Cantor's algorithm, it helps one to have exact formulae for the computations in Cantor's algorithm, i.e., a description of the algorithm in terms of only additions, multiplications and inversions in **k**. Explicit formulae have been completed for genus 2, 3 and 4 hyperelliptic curves. In genus 2 , the first work was done by [57], though improved methods have been found by Harley [22], Lange [29], Matsuo, Chao and Tsujii [34], Takahashi [58], and many others. In genus 3, Pelzl et al. [41] generalized work by Kuroki et al. [28] to give the first explicit formulae that work in all positive characteristic. The paper [41] also gives improvements on implementations of genus 2 hyperelliptic curves. A full description of genus 3 formulae can be found in Wollinger's PhD thesis [62]. For genus 4, Pelzl, Wollinger, and Paar gave the first explicit formulae in [42]. Their computations show that genus 4 arithmetic can compete with lower genus curves as far as computation costs are concerned.

8.7.2 Projective Coordinates

The operation of inverting elements in a finite field is much more costly than addition or multiplication. This has initiated research into finding ways to trade inversions for extra multiplications, additions, and storage in elliptic and hyperelliptic curve cryptography. If we wanted to compute nP for some element $P \in Jac(C)$ and $n \in \mathbf{Z}$ using the standard double-and-add method, we would be forced to use $O(\log n)$ inversions. By introducing another variable, Z, it is possible to delay performing inversions until the last step of the algorithm. For elliptic curves, this extra coordinate Z is equivalent to storing the point in projective coordinates. For higher genus this is no longer the case, but we still call these coordinates projective, due to the similarity with elliptic curves. Projective coordinates for elliptic curves will be covered in the following chapter of this book. Miyamoto et al. [39] first described

an algorithm for projective coordinates on genus 2 curves. This work has been improved by Lange [31] and others. Projective coordinates have also been described by Fan, Wollinger, and Wang [13] for genus 3 curves.

8.7.3 Other Optimization Techniques

Due to the rich structure of hyperelliptic curves, many other techniques exist for performing the group operation. Lange [30], has expanded upon projective coordinates for genus 2 curves in characteristic 2 to give better results. This work introduces several new variables to save on inversions, combining them to also save on other costs. Using special curves can also give remarkable speedups. Certain elliptic curves can be transferred to the Montgomery form, which aids considerably in computations. Gaudry has given a similar form for certain hyperelliptic curves in [19]. In Ref. [2] Bernstein and Lange showed that genus 2 hyperelliptic curves in Gaudry form can even outperform elliptic curves. The fact that hyperelliptic curves need smaller fields to obtain cryptographically secure group sizes is what potentially gives hyperelliptic curves the edge over elliptic curves. Koblitz curves provide another useful way of saving on implementation costs. A nice description of this theory for hyperelliptic curves can be found in [32], while the original idea can be found in Koblitz's work [27]. Additional methods for implementing the hyperelliptic curves will continue to be developed as cryptography in smaller, resource-constrained environments becomes necessary.

8.8 Counting Points on $Jac(C)$

In this section, we introduce methods for counting how many points there are on a given (hyper)elliptic curve over $\mathbf{k} = \mathbf{F}_q$. We begin with Schoof's method [50] for elliptic curves, which reduces the time taken from $O(q^{1/4+\varepsilon})$ to $O(\log^8(q))$. Refinements of this due to Elkies and Atkin [11], [1] reduce this further to $O(\log^6(q))$.

Let E be an elliptic curve defined over $\mathbf{F}_q$. By Hasse's theorem [53], the number of points n on it is $q+1-t$, where $|t| \leq 2\sqrt{q}$. Define ℓ_0 to be the smallest prime such that

$$\prod \ell > 4\sqrt{q}$$

where the product is over primes $\leq \ell_0$. Schoof's idea is to find $t \pmod{\ell}$ for all these primes, in which case the Chinese remainder theorem determines $t \pmod{\prod \ell}$ and so t (and hence n) exactly, since $\prod \ell$ is larger than the range t is confined to. By the Prime Number theorem, $\ell_0 = O(\log(q))$.

If $\ell = 2$, $n \equiv 1 \pmod{2}$ if and only if E has no point of order 2, which is easy to check. For example, if the curve has equation $Y^2 = f(X)$, then the points of order 2 correspond to roots of f, so $n \equiv 1 \pmod{2}$ if and only if f is irreducible over $\mathbf{F}_q$,

which happens if and only if $\gcd(f(X), X^q - X) = 1$. Likewise, for each ℓ, there is a polynomial $f_\ell(X)$ whose roots are the X-coordinates of points of order ℓ.

Suppose $\ell > 2$. We consider the Frobenius mapping ϕ on the points of E over $\bar{\mathbf{k}}$ defined by $\phi(x,y) = (x^q, y^q)$ and sending P_∞ to itself. The proof of Hasse's theorem establishes that

$$\phi^2(P) - t\phi(P) + qP = P_\infty$$

Let $q_\ell = q \pmod{\ell}$ and $t_\ell = t \pmod{\ell}$. For each $\tau \in \{0, 1, ..., \ell - 1\}$, we compute the x-coordinates of both $(x^{q^2}, y^{q^2}) + q_\ell$ and $\tau(x^q, y^q)$. Thanks to f_ℓ, these are both rational functions of x and y. Clearing denominators and using the equation of the curve to eliminate any nonlinear powers of y yields an equation of the form $a(x) - yb(x) = 0$. Substituting this into the curve equation produces a polynomial equation h just in x. Since we are seeking a point of order ℓ, all these calculations, so in particular h, can be taken $\pmod{f}_\ell$, which has degree $O(\ell^2)$.

To check if h has a solution that is a point of order ℓ, $\gcd(h, f_\ell)$ is computed. Only if it is nontrivial do we get a viable value of τ, i.e., $\tau = \pm t_\ell$. Either sign is possible since the x-coordinates are the same. A similar analysis of the y-coordinates determines which. This also means that τ only need run as far as $(\ell - 1)/2$. Most of the work is in computing $x^q, y^q, x^{q^2}, y^{q^2} \pmod{f_\ell}$ and, since f_ℓ is $O(\ell^2)$ and ℓ is $O(\log(q))$, the complexity is polynomial in $\log(q)$, namely $O(\log^8(q))$.

When $t^2 - 4q$ is a square $\pmod{\ell}$, the Frobenius map has an eigenvalue in $\mathbf{F}_\ell$, in which case a factor of degree $(\ell - 1)/2$ of f_ℓ can be used, as noted by Elkies. Atkin found a similar method in the case that $t^2 - 4q$ is not a square $\pmod{\ell}$, and together these yield the SEA (Schoof–Elkies–Atkin) algorithm with complexity $O(\log^6(q))$.

Now, let C be a hyperelliptic curve of genus g defined over $\mathbf{F}_q$. The theory of zeta functions tells us that there exist g complex numbers $\alpha_1, ..., \alpha_g$ with absolute value $\sqrt{q}$ such that, if N_r is the number of points on $C(\mathbf{F}_{q^r}$, then

$$N_r = q^r + 1 - \alpha_1^r - \overline{\alpha_1}^r - ... - \alpha_g^r - \overline{\alpha_g}^r$$

Note that when $g = 1$ and $t = \alpha_1 + \overline{\alpha_1}$, we get Hasse's inequality.

In general, $N_1, ..., N_g$ determine $\alpha_1, ..., \alpha_g$ (and so N_r for all r). They also determine the order of the group $Jac(C)$, which turns out to be $\prod_1^g (1 - \alpha_i)(1 - \overline{\alpha_i}) = \prod_1^g (1 + q - \alpha_i - \overline{\alpha_i})$.

Over finite fields with small characteristic, Satoh's p-adic approach [48] is asymptotically faster than the SEA algorithm. It was extended to Characteristic 2 by Skjernaa [55] and Fouquet, Gaudry, and Harley [14] with a memory-efficient version introduced by Vercauteren [61]. Mestre's AGM (arithmetic-geometric mean) method [37] gave the same asymptotic behavior but with a better constant, while Satoh, Skjernaa, and Taguchi [49] gave a quicker method if one allows precomputations.

As regards higher genus curves, Pila [43] gave an impractical generalization of Schoof's algorithm. Satoh's approach does not work well since the Serre–Tate canonical lift of the Jacobian need not be a Jacobian. Mestre's AGM method is only practical in genus ≤ 2.

This led to the introduction of new techniques. Kedlaya [25] used Washnitzer–Monsky cohomology to count points in small, odd characteristic in time

$$O(g^{4+\varepsilon}\log^{3+\varepsilon}(q)) .$$

This was extended to characteristic 2 by Denef and Vercauteren [7] and Vercauteren [60]. Using Dwork cohomology, Lauder and Wan [33] produced a practical method to count points on Artin–Schreier curves. Both these approaches take $(g^{5+\varepsilon}\log^{3+\varepsilon}(q))$ time.

8.9 Attacks

Having introduced elliptic and hyperelliptic curve cryptography, we now consider potential vulnerabilities of these systems. Over time, researchers have discovered several possible attacks on ECC and HCC that someone looking to implement these systems should be aware of. Avoiding them informs our choice of suitable (hyper)elliptic curve. In general, Shanks' baby-step giant-step method and Pollard's methods (see Sections 9.1 and 9.2) improve on sheer brute force attack by exploiting an idea called the birthday attack to solve discrete logarithm problems in any abelian group. These take on the order of $\sqrt{n}$ operations, where n is the size of the group (so about $q^{g/2}$ in the case of a curve of genus g over $\mathbf{F}_q$). Certain (hyper)elliptic curves are vulnerable to other methods, described later in this section, and any user of ECC or HCC should avoid this choice of curve.

8.9.1 Baby-Step Giant-Step Attack

Shanks' baby-step giant-step algorithm [3] works for any abelian group G. Let us assume G has prime order n (as is recommended by the results in the next subsection) and that we wish to solve the discrete logarithm problem $Q = mP$ for m. Write $m = a\lceil\sqrt{n}\rceil + b$ with $0 \le a,b < \lceil\sqrt{n}\rceil$. Then $Q - bP = a\lceil\sqrt{n}\rceil P$. We make a table of baby steps $Q - bP, b = 0,1,...,\lceil\sqrt{n}\rceil - 1$, and we then start computing giant steps $a\lceil\sqrt{n}\rceil P, a = 0,1,...,\lceil\sqrt{n}\rceil - 1$. Once we find a point that also occurred in one of our baby-step tables, we have found a,b and so have solved the DLP.

It should be noted that an exact value for n is not needed, just an upper bound – in fact the method can be adapted to yield n assuming G is cyclic [12]. A drawback of the baby-step giant-step method is the large amount of memory required if n is large (about $\sqrt{n}$ entries of length $\log n$).

8.9.2 Pollard Rho and Lambda Attacks

Pollard's rho method [45] avoids this by employing a single random walk that eventually self-intersects solving the problem (and that looks like the Greek letter

rho). Pollard's lambda method (or tame kangaroo/wild kangaroo method) uses two random walks, the goal again being to find a collision, but this time with the tame kangaroo laying traps for the wild kangaroo (so that they produce the Greek letter lambda). Various authors [4], [59] have experimented with the parameters of this method. In particular, van Oorschot and Wiener [40] provide a significant speedup by using several kangaroos in parallel.

The main idea is as follows. Let P, Q be points in G, an abelian group of order n, such that $Q = mP$. Let $f : G \to \{1, ..., s\}$ be a function equidistributed in the sense that

$$\sum_{i=1}^{s} || \{g \in G : f(g) = i\} \mid -n/s \mid = O(\sqrt{n})$$

Given a starting point $g_0 \in G$, we define a random walk $g_k = F(g_{k-1})$, where $F(g) = g + M_{f(g)}$. Here $M_i = a_i P + b_i Q$ for $i = 1, ..., s$ is a set of multipliers. Teske [59] found that s approximately 20 worked best. You set off two kangaroos performing these jumps.

Applying this to two kangaroos, you reach a collision so that $x_{i1}P + x_{i2}Q = y_{j1}P + y_{j2}Q$, so that $(x_{i1} - y_{j1})P = (y_{j2} - x_{i2})Q = (y_{j2} - xi2)mP$. Since $\gcd(x_{i1} - y_{j1}, n) = 1$, this can then be solved for m.

8.9.3 Pohlig–Hellman Attack

The Pohlig–Hellman algorithm [44] reduces the discrete logarithm problem in any abelian group to a subgroup of prime order. Thus, the order of the group we choose (the number of points on the elliptic curve or Jacobian of the hyperelliptic curve) should have a very large prime divisor and we will take our generating point P to be of that order.

To see this, suppose that the order of the group $n = \prod_{i=1}^{r} p_i^{k_i}$ and that we wish to solve the problem $Q = mP$ for m. Letting $n' = n/p_1^{k_1}$ and $m_1 = m \pmod{p_1}$, we can solve $Q' = n'Q = m_1 P'$ where $P' = n'P$ is a point of prime order p_1 to get m_1. Then $m_i = \pmod{p_1^i}, i = 2, 3, ...$ are successively computed as follows. Say m_i is known and $m = m_i + cp_1^i$. Then we know $Q - m_i P = cp_1^i P = cR$ and R, which has order $n_i = n/p_1^i$. So $c \pmod{p_1}$ is found and we have m_{i+1}. Once $m \pmod{p_i^{k_i}}$ is known for all i, the Chinese remainder theorem determines m.

8.9.4 Menezes–Okamoto–Vanstone Attack

The Menezes–Okamato–Vanstone (MOV) attack [35] uses the Weil pairing to embed the group of points on an elliptic curve over $\mathbf{F}_q$ in the multiplicative group of a larger finite field. If this finite field is only a small degree extension of $\mathbf{F}_q$, then we will be vulnerable to index calculus or number field sieve attacks on the discrete

logarithm problem in the multiplicative group of the extension field. This happens in particular with supersingular elliptic curves, where the degree is at the most 6. These curves were favored because addition in them involves fewer operations, but now should be avoided. For supersingular hyperelliptic curves of genus 2, the extension degree is at the most 30 [18].

The way the MOV attack works is as follows. Suppose we wish to solve $Q = mP$ for m, where because of the Pohlig–Hellman attack we are assuming P has prime order n. Let e be the smallest positive integer such that $q^e \equiv 1 \pmod{n}$. This ensures that $\mathbf{F}_{q^e}$ contains primitive nth roots of 1. For supersingular curves, Menezes [35] showed with case-by-case consideration that $e \leq 6$. Curves of trace 2 are also bad since $n = q + 1 - 2 = q - 1$ and so $e = 1$.

Say $e > 1$. The Weil pairing is a pairing

$$E(\mathbf{F}_{q^e})/nE(\mathbf{F}_{q^e}) \times E[n] \to \mathbf{F}_{q^e}^*/(\mathbf{F}_{q^e}^*)^n$$

This injects $< P >$ into a subgroup of $\mathbf{F}_{q^e}^*/(\mathbf{F}_{q^e}^*)^n$ and so maps the discrete logarithm problem over to the multiplicative group of a finite field, where subexponential methods can be used if e is reasonably small. This includes index calculus methods where a set of elements is chosen to act as a factor base. Enge [12] has a form of index calculus for attacking elliptic curve cryptosystems directly, but these are ineffective for large field sizes. The case $e = 1$ is similar but runs into a small technicality involving non-degeneracy of the Weil pairing [3]. Frey and Rück [16] introduced a similar method that uses the Tate pairing.

8.9.5 Semaev, Satoh-Araki, Smart Attack

An anomalous elliptic curve is one with exactly q points. The Semaev, Satoh-Araki, and Smart attack [51], [47], [56] maps the group to the additive group of $\mathbf{F}_q$, yielding a polynomial-time attack (the other attacks listed here are at best subexponential). The main idea is as follows. Suppose we wish to solve $Q = mP$ in the elliptic curve E over $\mathbf{F}_q$. There is a unique smallest complete local ring of characteristic zero, $\mathbf{Z}_q$, the q-adic integers, and by Hensel's lemma we can lift P and Q to points defined over $\mathbf{Z}_q$, say $\tilde{P}$ and $\tilde{Q}$. Then $qE(\mathbf{Z}_q)/q^2E(\mathbf{Z}_q) \cong \mathbf{F}_q$ and denoting this map by log we have that $\log(q\tilde{P}) = m\log(q\tilde{Q})$. Then solving the discrete logarithm problem in the additive group of a field is trivial, using Euclid to invert $\log(q\tilde{Q})$.

8.9.6 Attacks employing Weil descent

More recently, Weil descent has been used for some special finite fields. In 2000, Gaudry, Hess, and Smart [20], extending the work of Frey [15] and Galbraith [17], showed how to reduce a discrete logarithm problem in $E(\mathbf{F}_{q^e})$ to a discrete logarithm

problem in $\mathrm{Jac}(C)(\mathbf{F_q})$ where C is a hyperelliptic curve. The idea is to use the Weil restriction of E, which is an e-dimensional abelian variety over $\mathbf{F}_q$, and intersect it with $e-1$ hyperplanes to obtain C. For example, if $q = 2^{31}$ and $e = 5$, we obtain a curve C of genus at most 16, so it is possible to attack many elliptic curves over 2^{155} this way. The GHS attack has yet to be shown to be effective in practice – in particular none of the ten elliptic curves in the standards is vulnerable to it. Thanks to the work of Gaudry, and later Diem [8], elliptic curves over fields $\mathbf{F}_{p^n}$, where both p and n are large, are vulnerable to this method. Diem and Thomé [9] have also introduced an index calculus method for non-hyperelliptic curves of genus 3.

8.10 Good Curves

Putting together the attacks from the previous section leads to design criteria for the underlying elliptic or hyperelliptic curve. To set up an ECC, we should use an elliptic curve over $\mathbf{F}_q$ and a subgroup of order n where:

(i) n should be prime (Pohlig-Hellman);
(ii) q should be of the order of 1000 bits to be truly considered secure, but in practice 160 bits is considered equivalent to about 1024-bit RSA and 190 bits to 2048-bit RSA. Thanks to baby-step giant-step and Pollard's methods, these are considered equivalent to 80-bit and 95-bit symmetric cryptosystems respectively;
(iii) the curve should not be anomalous, so n should not equal q (Semaev, Satoh-Araki, Smart);
(iv) the smallest positive integer e such that $q^e \equiv 1 \pmod{n}$ should be large so the curve should not be of trace zero or two nor supersingular (Menezes–Okamoto–Vanstone);
(v) certain ground fields, e.g., $\mathbf{F}_{2^{155}}$, should be avoided (Weil descent).

The standards provide suitable curves. For example, FIPS 186-2 lists 10 fields and methods to choose elliptic curves that will produce secure cryptosystems.

8.11 Exercises

1. Prove that for an elliptic curve, the "chord and tangent" addition rule described in Section 8.6 is the same as the one given in Cantor's algorithm.
2. If the characteristic of $\mathbf{k}$ is not 2, show that any hyperelliptic curve C_1 with equation:
$$Y^2 + h_1(X)Y = f_1(X)$$
is isomorphic to a hyperelliptic curve C_2 with equation:
$$Y^2 = f_2(X)$$
An isomorphism between hyperelliptc curves is a linear map $(X,Y) \to (aX + bY, cX + dY)$ such that the points on C_1 are mapped to the points on C_2.

3. Let E be the elliptic curve $y^2 = x^3 + 81x + 103$ defined over $\mathbf{F}_{1013}$. Show that it has 962 points. Since 962 factors as $2 \times 13 \times 37$, you can use Pohlig–Hellman to solve the following discrete logarithm problem. Let $P = (1,728)$ and $Q = (769,175)$. Find an integer m such that $Q = mP$.
4. Let E be the same elliptic curve as in Question 3. Find the smallest positive integer e (embedding degree) such that with $q = 1013$ and $n = 962$, $q^e \equiv 1 \pmod{n}$. Convert the discrete logarithm problem of question 1 into one in $\mathbf{F}_{q^e}$ and estimate how long it will take to solve using index calculus methods.
5. Let E be the elliptic curve $y^2 = x^3 + 141x + 30$ defined over $\mathbf{F}_{1013}$. Show that this curve is anomalous. Construct an explicit isomorphism between this and the additive group of $\mathbf{F}_{1013}$. Using this, if $P = (1,292)$ and $Q = (316,412)$, find an integer m such that $Q = mP$.
6. Suppose E is an elliptic curve with equation $y^2 = x^3 + ax$ over prime field $\mathbf{F}_p$ with $p \equiv 3 \pmod{4}$. Show that it has exactly $p + 1$ points. [Hint: how many points do x and $-x$ together contribute?]. What is the embedding degree? Show how one can map arbitrary ID-strings to points on $E(\mathbf{F}_p)$ (so that it can be used in identity-based cryptography).

8.12 Projects

1. To show why Elliptic Curve Cryptography has had such an impact, implement both ECC and RSA and compare their timings. For details on RSA, consult [46]. Remember that for similar security, a key size of 160 bits in ECC is equivalent to 1024-bit RSA. Thus, even if the speed of RSA is better, transmission and storage costs for ECC are lower. For previous results on RSA vs ECC for 8-bit processors, see [21].
2. Consider the curve $x^2 + y^2 = a^2(1 + x^2y^2)$, where $a^5 \neq a$. Show that this defines an elliptic curve and that every elliptic curve is, possibly over an extension field, isomorphic to such a curve. Figure out explicit rules for addition and point-doubling. A very recent article by Harold Edwards in the Bulletin of the AMS (July 2007) carries out these calculations and more. Bernstein and Lange have suggested that this form could be very good for ECC, ensuring it is better than genus 2 HCC. Test out this claim in practical implementations, comparing this ECC with state-of-the-art HCC as found in papers on Gaudry's and Lange's websites.

References

1. A. O. L. Atkin. *The number of points on an elliptic curve modulo a prime*, Series of emails to the NMBRTHRY mailing list, 1992
2. D. J. Bernstein and T. Lange. *Elliptic vs. hyperelliptic*, (parts 1 and 2), talks at ECC-06

3. I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*, London Mathematical Society Lecture Note Series, Cambridge University Press, 1999
4. I. Blake, G. Seroussi, and N. Smart. *Advances in Elliptic Curve Cryptography*, London Mathematical Society Lecture Note Series, Cambridge University Press, 2004
5. D. G. Cantor. Computing in the Jacobian of a hyperelliptic curve. In: *Mathematics of Computation*, 48(177): 95–101, 1987
6. H. Cohen. A Course in Computational Algebraic Number Theory, *Graduate Texts in Mathematics* 138, 1993
7. J. Denef and F. Vercauteren. An extension of Kedlaya's algorithm to Artin-Schreier curves in characteristic 2, in ANTS-V, 2002
8. C. Diem. The GHS attack in odd characteristic, *Journal of Ramanujan Mathematical Society* 18(1): 1–32, 2003
9. C. Diem and E. Thomé. "Index calculus attacks in class groups of non-hyperelliptic curves of genus three", *Journal of Mathematical Cryptology* 2, to appear, 2008
10. W. Diffie and M. E. Hellman. New directions in cryptography, IEEE Transaction Information Theory, IT-22, 6: 644–654, 1976
11. N. Elkies. Elliptic and modular curves over finite fields and related computational issues In: *Computational Perspectives on Number Theory*, 21–76, 1998
12. A. Enge. *Elliptic Curves and Their Applications to Cryptography, An Introduction*, Kluwer Academic Publishers 1999
13. X. Fan, T. Wollinger, and Y. Wang. Inversion-Free Arithmetic on Genus 3 Hyperelliptic Curves and Its Implementations, *International Conference on Information Technology: Coding and Computing - ITCC*, April 11–13, 2005
14. M. Fouquet, P. Gaudry, and R. Harley. On Satoh's algorithm and its implementation, *Journal of Ramanujan Mathematical Society* 15: 281–318, 2000
15. G. Frey. *How to disguise an elliptic curve*, talk at ECC '98, 1998
16. G. Frey and H. Rück. A remark concerning *m*-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of Computation*, 62(206): 865–874 (1994)
17. S. Galbraith. Limitations of constructive Weil descent. In: *Public-Key Cryptography and Computational Number Theory*, 59–70, de Gruyter, 2000
18. S. Galbraith. "Supersingular curves in cryptography", *LNCS* 2248: 200-217, 2002
19. P. Gaudry. Fast genus 2 arithmetic based on theta functions, *Journal of Mathematical Cryptology*, 1: 243–266, 2007
20. P. Gaudry. F. Hess, and N. Smart, Constructive and destructive facets of Weil descent on elliptic curves. *Journal of Mathematical Cryptology*, 2000
21. N. Gura. A. Patel, A. Wander, H. Eberle, and S. Shantz, *Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs*, CHES2004, Cambridge (Boston), 2004
22. R. Harley. *Fast Arithmetic on Genus Two Curves*, http://cristal.inria.fr/∼harley/hyper/, (2000)

23. M. Jacobson, N. Koblitz, J. Silverman, A. Stein, and E. Teske. Analysis of the xedni calculus attack. *Designs, Codes, and Cryptography*, 20(1): 41–64, 2000
24. M. Jacobson, A. Menezes, and A. Stein. "Solving elliptic curve discrete logarithm problems using Weil descent", *Journal of Ramanujan Mathematical Society* 16(3): 231–260, 2001
25. K. Kedlaya. "Counting points on hyperelliptic curves using Monsky-Washnitzer cohomology", *Journal of Ramanujan Mathematical Society* 16: 323–338, 2001
26. N. Koblitz. Elliptic curve cryptosystems. In: *Mathematics of Computation* 48: 203–209, 1987
27. N. Koblitz. Hyperelliptic cryptosystems. *Journal of Mathematical Cryptology* 1: 139–150, 1989
28. J. Kuroki, M. Gonda, K. Matsuo, J. Chao, and S. Tsujii. Fast Genus Three Hyperelliptic Curve Cryptosystems. In *Proceedings of SCIS*, 2002
29. T. Lange. *Efficient Arithmetic on Hyperelliptic Curves*, PhD Thesis. Universitat-Gesamthochschule Essen, 2001
30. T. Lange. Weighted Coordinates on Genus 2 Hyperelliptic Curves. *Cryptology ePrint Archive*, Report 2002/153, 2002
31. T. Lange. Inversion-Free Arithmetic on Genus 2 Hyperelliptic Curves. Preprint, 2002
32. T. Lange, C. Günther, and A. Stein. Speeding up the arithmetic on hyperelliptic Koblitz curves of genus 2, SAC 2001, LNCS 2012, Springer 106–117, 2001
33. A. Lauder and D. Wan. Computing zeta functions of Artin-Schreier curves over finite fields, London Math Soc. JCM 5: 34–55, 2002
34. K. Matsuo J. Chao, and S. Tsujii. Fast Genus Two Hyperelliptic Curve Cryptosystems, *Proc. Second Int'l Symp. Electronic Commerce* (ISEC 2001), 2001
35. A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to a finite field. *IEEE Transaction on Information Theory*, 39: 1639-1646, 1993
36. A. Menezes, Y-H. Wu, and R. Zuccherato. *An Elementary Introduction to Hyperelliptic Curves*. Technical Report CORR 96-19, Department of Combinatorics and Optimization, University of Waterloo, Ontario, Canada, (1996)
37. J. F. Mestre. *AGM pour le genre 1 et 2*, lettre à Gaudry et Harley, Dec 2000
38. V. Miller. Use of elliptic curves in cryptography, *CRYPTO* 85, 1985
39. Y. Miyamoto, H. Doi, K. Matsuo, J. Chao, and S. Tsuji. A Fast Addition Algorithm of Genus Two Hyperelliptic Curve, *Proceedings of SCIS 2002*, 497-502, in Japanese, 2002
40. P.van Oorschot and M. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Mathematical Cryptology*, 12, no. 1, 1-28 1999
41. J. Pelzl, T. Wollinger, J. Guajardo, and C. Paar. Hyperelliptic curves cryptosystems: closing the performance gap to elliptic curves. *Cryptology ePrint Archive*, 2003, http://eprint.iacr.org/

42. J. Pelzl, T. Wollinger, and C. Paar. Low Cost Security: Explicit Formulae for Genus-4 Hyperelliptic Curves, In Tenth Annual Workshop on Selected Areas in Cryptography, 2003
43. J. Pila. Frobenius maps of abelian varieties and finding roots of unity in finite fields. *Mathematics of Computation* 55: 745-763, 1990
44. G. Pohlig and M. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Transaction on Information Theory*, 24: 106-110, 1978
45. J. Pollard. Monte Carlo methods for index computation mod p. *Mathematics of Computation*: 918-924 (1978)
46. R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21 (2): 120-126, 1978
47. T. Satoh and K. Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves, Comm. Math. Univ. Sancti Pauli, 47(1): 81-92, 1998
48. T. Satoh. The canonical lift of an ordinary elliptic curve over a finite field and its point counting, *Journal of Ramanujan Mathematical Society*. 15: 247-270, 2000
49. T. Satoh, B. Skjernaa, and Y. Taguchi. Fast computation of canonical lifts of elliptic curves and its application to point counting, *Finite Fields and Their Applications* 9: 89-101, 2003
50. R. Schoof. Elliptic curves over finite fields and the computation of square roots mod p, *Mathematics of Computation* 44: 483-494, 1985
51. I. A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p, 67(221): 353-356, 1998
52. V. Shoup. Lower bounds for discrete logarithms and related problems. In *Proc. Eurocrypt* '97, pp. 256-266, 1997
53. J. H. Silverman. The arithmetic of elliptic curves. *Graduate Texts in Mathematics*, vol 106, Springer-Verlag, 1986
54. J. H. Silverman. The xedni calculus and the elliptic curve discrete logarithm problem. *Designs, Codes, and Cryptography*, 20: 5–40, 2000
55. B. Skjernaa. Satoh's algorithm in characteristic 2. *Mathematics of Computation* 72: 477–488, 2003
56. N. Smart. The discrete logarithm on elliptic curves of trace one. *Journal of Mathematical Cryptology*, 12: 193–206, 1999
57. A. M. Spallek. *Kurven vom Geschlecht 2 und ihre Anwendung in Public-Key-Kryptosystemen*, PhD Thesis. Universitat Gesamthochschule Essen, 1994
58. M. Takahashi. *Improving Harley Algorithms for Jacobians of Genus 2 Hyperelliptic Curves*, In SCIS, IEICE Japan, 2002. in Japanese.
59. E. Teske. Speeding up Pollard's rho method for computing discrete logarithms. *LNCS*, 1423: 541–554, 1998
60. F. Vercauteren. Computing zeta functions of hyperelliptic curves over finite fields of characteristic 2. In "Advances in cryptology - CRYPTO 2002", *LNCS* 2442: 369–384, 2002

61. F. Vercauteren, B. Preneel, and J. Vandewalle, A memory efficient version of Satoh's algorithm. In "Advances in Cryptology - EUROCRYPT 2001", *LNCS* 2045, 1–13 (2001)
62. T. Wollinger. *Software and Hardware Implementation of Hyperelliptic Curve Cryptosystems*, Ph.D. Thesis, Ruhr-Universitt Bochum, Germany, July 2004

Chapter 9
Instruction Set Extensions for Cryptographic Applications

Sandro Bartolini, Roberto Giorgi, and Enrico Martinelli

9.1 Introduction

Instruction-set extension (ISE) has been widely studied as a means toimprove the performance of microprocessor devices running cryptographic applications. It consists, essentially, in endowing an existing processor with a set of additional instructions that can be useful for speeding up specific cryptographic computations. Recently, researchers became aware of the following: "The efficiency of an implementation algorithm often depends heavily on the details of the target platform, *e.g.*, on the instruction set or the pipeline of a processor. Hence, theoretical complexity measures, such as the bit complexity, can be misleading in practice" ([47]).

In this chapter, we will analyze the implications of designing and deploying an ISE for a microprocessor. We will give details on existing research proposals for various cryptographic applications, highlighting the associated benefits and limitations, and we will show the ISEs that are available in some market products and are proposed in research studies.

9.1.1 Instruction Set Architecture

Instruction-set extension can be better understood only after having a clear idea of what an instruction-set is. At the higher level, an instruction-set (or instruction-set architecture – ISA) can be defined as the pool of instructions made available by a processor to the assembler programmer, or to the compiler.

In this sense, the ISA defines a significant quote of the programming interface of the processor: the basic operations that the outside world can ask the processor to do. The whole programming interface of a processor is surely wider than the sole ISA and, in brief, it encompasses also the structure and features of the processor

Università di Siena, Italy,
e-mail: bartolini,giorgi,enrico@dii.unisi.it

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_9,

© Springer Science+Business Media, LLC 2009

registers, the organization of the memory space, as it is perceived by the programmer (*e.g.*, virtual memory, permissions), and the features of the I/O (input/output) space. All this information is needed to take full advantage from the features exposed by a microprocessor.

Anyway, instructions cannot be arbitrarily complex: a trade-off has to be done in order to have fast circuits behind the implementation, and the RISC approach (simple, modular, efficient instructions, *e.g.*, MIPS, SPARC, PowerPC) versus the CISC approach (many powerful instructions, *e.g.* Intel x86) has characterized the main microprocessors on the market.

For instance, we will briefly outline some of the features of the ISA of Intel® processors and its evolution through specific extensions during the years.

The base ISA of the Intel Pentium-4 class processors [31] is almost the same since the old 386-class processor and is named x86 instruction set. It comprises hundreds of instructions, operating on eight 32-bit general-purpose registers. They can be seen also as the sole 16-bit lower part and four of them even allow using their 16-bit lower part as two 8-bit registers (*e.g.*, *AH* and *AL* are 8-bit registers that, together, form the *AX* 16-bit register, which is the lower part of the 32-bit register *EAX*). Specific instructions for integer arithmetic, bitwise operations, movement among registers, and between registers and memory or I/O space can use 8-, 16-, or 32-bit registers. Other instructions manage the program control flow at various levels: jumps, calls and loops, up to interrupt service routine management. This 32-bit ISA has been extended to a 64-bit backward compatible ISA (x86-64) in 2004, after that a similar proposal was done in 2002 by AMD. This extension was motivated by the need to access wide memory regions (*i.e.*, beyond 2–4 GB, according to the available operating system) easily, and to support an increased word-level parallelism which was needed by a number of high-end applications.

Since some versions of the 486 model, the processor was extended to natively support floating-point operations, without the need of an external coprocessor. In this way, the programmer sees some additional configuration registers and eight 80-bit registers for working on floating-point operands. Specific instructions move the operands to/from the floating-point register set and trigger floating-point computations. Specific circuits implement the register file and the operations that are performed upon instruction execution.

Another class of extensions have been proposed, in steps, for vector-like operations which are motivated by the need of supporting efficiently a variety of multimedia applications such as 2-D and 3-D graphics, motion video, image processing, speech recognition, audio synthesis, telephony, and video conferencing. Beginning with the Pentium II and Pentium with Intel MMX technology processor families, a number of incremental extensions have been introduced into the IA-32 architecture to permit IA-32 processors to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE, SSE2, SSE3, and SSE4 extensions. Each of these extensions provides a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements contained in specific registers (64-bit MMX or 128-bit XMM registers).

Multimedia extension allows the programmer to see eight 64-bit registers which can be used as groups of eight packed bytes, four packed 16-bit words, or two packed 32-bit doublewords. In this way, specific instructions can operate on such vector-like operands more efficiently (*e.g.*, parallel saturating addition on all packed elements) and other instructions are provided for loading/storing values from/to MMX registers. Architecturally, specific circuits for parallel elaboration of SIMD operations are added to the processor, while *the MMX register file is shared with the floating-point unit*. In this way, no additional storage for MMX registers was needed but, as a drawback, great attention has to be taken when using in the same time both floating-point and MMX instructions.

The SSE extension introduces a separate set of eight 128-bit registers (*XMM*) for SIMD operations, which are intended to support floating-point SIMD operations too. With a dedicated register file, the conflicts with other internal resources, as in the case of the floating-point register file used by MMX, are reduced. In particular, each *XMM* register can be seen by the programmer as four 32-bit single-precision floating-point values. The SIMD operations on such values can help in supporting advanced media and communications applications, for which MMX integer/fixed-point SIMD operations are limiting. Move and conversion instructions too are provided for the interaction of *XMM* registers with memory, MMX and general-purpose registers.

The SSE2 extension increases flexibility in using *XMM* registers as additional floating-point packed values, and introduces the support for packed integers too. In fact, each *XMM* register can be used also as two packed 64-bit double-precision floating-point values, 16 packed bytes, eight packed 16-bit words, four packed 32-bit doublewords, and two packed 64-bit quadwords. Operations on packed integers into SSE registers allow double parallelism than using MMX and avoid conflicts with the floating-point unit. Additional instructions are provided to operate on this variety of operand types.

The SSE3 extension enhances the previous instruction set with only 13 instructions that accelerate some SSE, SSE2, and floating-point capabilities. For instance, an optimized floating-point to integer conversion instruction is provided, as well as an unaligned 128-bit load instruction for integer operands, and additional SIMD operations. This highlights the importance of easying the interaction between the existing processor data formats (*e.g.*, integers and float values) and hardware modules from one side, and the extended circuitry (*e.g.*, registers) and operands (*e.g.*, float or integer packed values) on the other side, in order to boost programmability and performance.

The SSE4 [32] extension was recently proposed with the Intel® Core™ processor family and further improves SSE capabilities. Essentially, it improves the flexibility of SSE in supporting compiler vectorization and significantly increases the available packed doubleword computations.

This brief story of ISA extensions in Intel® processors highlights that the study of the interaction between special hardware resources (*e.g.*, registers, circuits) and the set of available instructions is crucial for accelerating specific computations and thus for the final performance of an ISA extension.

It is very interesting to highlight here the implications of ISA extensions towards an operating system (OS). This occurs mainly when an OS, even minimal, is employed to enable multiprogramming through processor virtualization. In fact, in a multiprogrammed system, the physical processor is *assigned* by the OS to the different processes (*i.e.*, running programs) that are contemporarily in execution so that each one is able to execute, even if in an intermittent fashion. The OS switches the process–processor association according to the time spent by the running process (*e.g.*, round-robin) or when the latter blocks(*e.g.*, wait for an external event). When a process $P1$ leaves the processor to the next process $P2$ (*i.e.*, a context switch happens), the complete *processor state* has to be saved so that it can be restored later when $P1$ will be able to continue its execution exactly as if it was never interrupted at all.

The state of a process comprises, at least, the value of all processor registers, including the state registers (*e.g.*, flags). Therefore, extending the register organization of a processor implies modifications into the OSs in order to properly manage the machine state. If the OS is not updated upon a context switch, it saves only the original processor state and it neglects the additional extended registers. In this way, when the process state will eventually be restored on the processor, a part of its state can be corrupted and the elaboration can become erroneous. This might be taken into account when designing a solution based on ISE for a target processor for which a number of OSs are already present in the market. All of them have to be updated to support certain ISA extensions.

On this point, note that the Intel MMX extension could be supported without OS modifications because they relied on the registers already used by the floating-point unit.

In the following sections, we will analyze various proposals for ISEs for cryptographic application, highlighting, where possible, the motivation for the proposal, the hardware and software features, as well as the resulting performance benefits.

9.2 Applications and Benchmarks

It has become quite clear that a successful cryptographic system needs an understanding of the applications that it will run. Moreover, its performance strongly depends on an efficient implementation of its essential operations [47].

Therefore, possible ISEs also depend on the applications that we currently focus on. At higher level, secure IP (IPSEC), virtual private networks (VPNs), just to give some examples, are further emphasizing the importance of cryptographic processing among all types of communications. Also, mobile (cellular phone) systems, like 3G and beyond, will be using secure methods for payments, digital rights management, handset, and network authentication.

The above scenarios imply that ISEs will have to consider: (i) appropriate benchmarks in order to analyze the real demand for new instructions to be supported; (ii) appropriate platforms (high performance versus embedded systems) for considering the ISE.

9.2.1 Benchmarks

Current research [1, 6, 11, 16, 22, 23, 34, 39, 57] is essentially considering as benchmarks the most-used ciphers for both private-key (3DES, Blowfish, IDEA, Mars, RC4, RC6, AES) and private-key (RSA, DSA, Diffie-Hellman, El-Gamal, and their ECC variants).

Benchmark suites, which also highly condition the research on ISE and the choice of possible platforms, normally include some applications in the security domain. For example, MiBench [24] includes both applications like PGP [60], the famous encryption algorithm developed by Phil Zimmermann (relying on RSA or DSA), ciphers like Blowfish [8], Rijndael/AES [45] and hash calculations like SHA [46] (used in the well-known MD4 and MD5 hashing functions). Another effort to provide a reference benchmark suite for security is the Basicrypt benchmark package [3], which contains standard and elliptic curve code for Diffie-Hellman key exchange, digital signature algorithm (DSA), El-Gamal, and RSA encryption/decryption. Standard algorithms can be used with various key lengths (1024, 2048, and 3072), while for elliptic curve variants parameter files are defined according to fields and curves recommended by NIST standard FIPS 186-2. Public-key benchmarks in Basicrypt package were written using MIRACL C [53] procedures for big integer arithmetic. The MIRACL library consists of over 100 routines that cover all aspects of multiprecision arithmetic and offer procedures for finite-field elliptic curve operations.

Other approaches [23, 34] try to derive the most important operations for the ISE directly from an algorithm implementation.

The two most notable ciphers for private-key cryptography are 3DES and AES as they are selected as US encryption standards. More recently, AES tends to replace 3DES in many applications; we will discuss AES ISEs more in detail in Section 9.3.2.

For public-key cryptography, RSA continues to be a leader in the implementation while ECC-based applications are gaining consensus thanks again to the NIST standardization. RSA is essentially based on modular exponentiation and basic operations are common to other cryptosystems such as ECC. ECC ISEs will be the object of Section 9.3.3.

9.2.2 Potential Performance

Many of the above symmetric ciphers have little parallelism and few bottlenecks. For example, Blowfish, 3DES, IDEA, and RC6 can run within 20% of the performance of a pure dataflow machine [57]. There is more headroom for Mars and Twofish with potential speedups of 29% and 76% respectively. RC4 and AES have much more parallelism and could be sped up with more capable hardware [57]. Similar studies have been performed for ECC in the work of Bartolini et al. [1]. There are potentials for speeding up ECC, especially considering ISE for polynomial multiplication.

9.3 ISE for Cryptographic Applications

In the following sections we will analyze possible ISEs for cryptographic applications. In particular, each section goes into the details of the extensions specifically proposed for information confusion and diffusion, AES symmetric block-cipher and elliptic-curve public-key cryptosystem.

9.3.1 Instructions for Information Confusion and Diffusion

A common operation, especially in symmetric-key algorithms, consists in breaking a message into blocks and using confusion and diffusion to manipulate blocks of plaintext and transform it into ciphertext [50]. The goal of confusion is to obscure the relationship between plaintext and ciphertext by, for example, permuting certain bits. Cypher algorithms, such as Twofish, employ a series of reversible operations to implement a process called diffusion. The goal of diffusion is to impress upon each of the output bits some information from each of the input bits. The diffusion process is also conditioned by the private key, thus constituting an important step for increasing the resistance to ciphertext attackers.

Typical operations to support confusion and diffusion consist in permutations, rotates, substitutions. According to [57], the most common operations of cryptographic kernels such as 3DES, Mars, RC4, Rijndael, Twofish, besides other general operations of typical programs (such as arithmetic, logical, branches, and memory operations) are indeed: permutations, rotates, substitutions.

Rotates are easily reversible by rotating the same distance in the opposite direction and also have good diffusion properties.

The substitution operation can be implemented as a key-based transformation function using a byte-indexed array called an "SBOX", as in the Cryptomaniac processor [6]. Figure 9.1 illustrates the semantic of an SBOX.

Permutation operations rearrange the bits using a parametrized wire network called an "XBOX" in the work of [6]. Permutation is very effective in achieving diffusion [52] and is potentially very powerful and more general than rotates. In fact, an arbitrary permutation can achieve any one of $n!$ outcomes rather than one of n outcomes produced by a rotation.

Arbitrary permutation circuits are usually considered complex and have therefore been avoided in some algorithms such as Rijndael, RC5, RC6, IDEA, Mars, Kasumi. More recently [27], designs that propose to modify an existing shifter in order to perform both shift and also advanced operations to ease the permutations have been presented.

Another operation that has particularly good diffusion properties is modular multiplication [36]. This operation can be easily reversed with modular multiplication of the modular inverse. Also, modular addition has relatively good diffusion properties and it is easily reversed with modular subtraction. Modular operations will be analyzed in more detail in Section 9.3.3

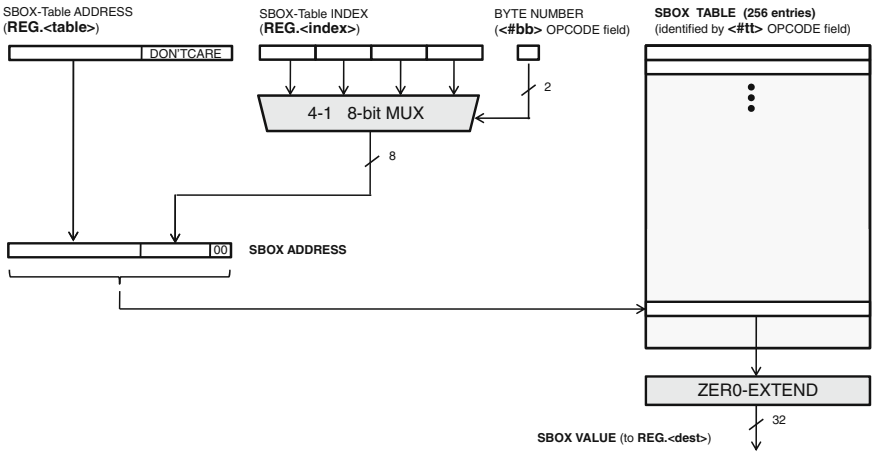

Fig. 9.1 SBOX semantics from Burke et al. [6].

9.3.1.1 Proposed Instructions

Existing ISAs mostly provide support for basic rotate operations, e.g., Intel, Alpha, SPARC, but lack support for permutations and substitutions.

A few ISAs such as PA-RISC [37] and IA-64 [30] support more advanced operations such as extract, deposit and mix, discussed in detail later in this section.

In the work of Burke et al. [6], some potentially useful instructions have been identified, in particular for substitutions (SBOX) and permutations (XBOX). Their methodology is also interesting: starting from commonly used cipher algorithms (such as 3DES, Blowfish, IDEA, Mars, RC4, RC6, Rijndael, Twofish), bottlenecks are identified, and an average of about 45% of dynamic instructions are identified as rotates, substitutions, and permutations.

Several instructions have been proposed in that work [6] and in a subsequent paper describing the Cryptomaniac processor [57], a flexible architecture for secure communications. Some of the proposed instructions are:

- `ROLX <src>, #<rot>, <dest>`
- `RORX <src>, #<rot>, <dest>`
  these two instructions respectively perform a rotate left and rotate right for the specified (`#<rot>`) number of bits of the source register (`<src>`), and a final XOR with another register `<dest>`. The result replaces the second register input `<dest>`. Such instructions are useful to speed up Mars and RC6. The timing analysis performed by the authors of this work indicated that such rotates easily fit in the cycle time of a small-sized ALU (arithmetic logic unit).

- `SBOX.#<tt>.#<bb>.<aliased> <table>,<index>,<dest>`

extracts byte #<bb> (0..3) from register <index>and concatenates the resulting 8-bit value with register <table> to produce a 32-bit aligned address to point an SBOX-table (see Figure 9.1). The 32-bit value from the SBOX-table is zero-extended and loaded into register <dest>. To speed up most SBOX operations, stores to SBOX-table are not visible by later SBOX instructions until an SBOXSYNC instruction is executed, unless the <aliased> flag is indicated. The instruction operates on a table identified by the table designator #<tt> (useful for a subsequent SBOXSYNC instruction). SBOX implementations take advantage of SBOX caches and other quite complex implementation details, making the implementation quite costly. Anyway, SBOX produces great benefits for algorithms such as Rijndael (AES), almost doubling the performance of this algorithm; in particular, having support in hardware for SBOX, reduces the latency for SBOX-table accesses from three instructions to one and speeds up from five cycles to two.

- SBOXSYNC.#<tt>
  synchronizes SBOX-table #<tt> with memory. This eliminates the need for SBOX instructions to snoop on store values in the processor core.

- XBOX.<bbb> <srca>, <srcb>, <dest>
  performs a partial general permutation of register <srca>, given the bit permutation map in register <srcb>; the result of the permutation is placed in register <dest>. The permutation map describes where each input operand bit is written in the destination and contains eight 6-bit indices to address each of the 64-bit <srca> register bits. The XBOX instruction opcode indicates through <bbb>, which of the weight bytes in the destination register are permuted. When XBOX is used, the 32-bit permutations in, e.g., 3DES are completed in 7 instructions (and executed in 3 cycles), yielding a significant improvement over the baseline code which requires 39 instructions.

Performance analysis done in the work of Burke [6] indicated a performance improvement when using the proposed instructions (plus a modular multiplication with modulus 33, and some processor refinements, i.e., doubling execution resources of a baseline aggressive superscalar processor such as Alpha). The improvement can achieve a 59% speedup over machines with basic rotate instructions and 74% speedup over machines without rotates.

Applications that take advantage from the speedup of this algorithm include web servers relying on SSL (or TLS) protocols, disk encryption/decryption, secure network using secure protocols such as IPSEC, and virtual private networks (VPNs).

In emerging applications such as cryptography (but also imaging and biometrics) more advanced bit-manipulation instructions are needed.

In the work of Lee et al. [27, 52], several bit-manipulation instructions are proposed:

- `EXTR r1=r2, <pos>, <len>`
- `EXTR.U r1=r2, <pos>, <len>`
  extracts and right justifies a single field from `r2` of bit length `<len>` from bit position `<pos>`; the high-order bits are filled with the sign bit of the extracted field (`EXTR`) or zero-filled (`EXTR.U`) as in Figure 9.2(a)) and the result is left in `r1`.

- `DEP.Z r1=r2, <pos>, <len>`
- `DEP r1=r2. r3, <pos>, <len>`
  deposits at bit position `<pos>` of single right-justified field from `r2` of bit length `<len>`; remaining bits are zero-filled (`DEP.Z`) or merged from second source register `r3` (`DEP`) as in Figure 9.2(b)).

- `MIX.{r,l}.{0,1,2,3,4,5} r1=r2, r3`
  selects right or left subword from pair of subwords, alternating between source registers r2 and r3; subword sizes are 2^i bits for i=0,1,2, ... ,5 for a 64-bit processor (see Figure 9.3).

A `MIX` operation is implemented in the PA-RISC [38] and IA-64 [30].

Another important class of operations for cryptography is permute [27, 52]. In most current ISAs, permute can be done using logical operations or table lookups (see `XBOX` mentioned earlier). This method may suffer memory latencies and cache misses. A generic n-bit permutation takes $O(n)$ instructions [52]. In the work of Shi et al. [51], the following simple permute instruction – named 'group' – is proposed:

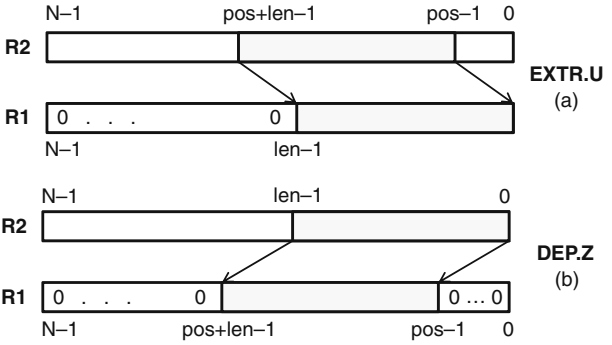

Fig. 9.2 (a) `EXTR.U r1=r2,<pos>,<len>; (b) DEP.Z r1=r2,<pos>,<len>` from Hilewitz et al. [27].

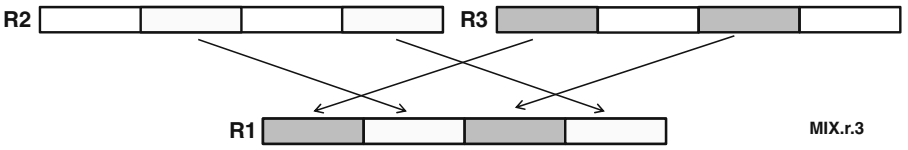

Fig. 9.3 Mix right operation, from Hilewitz et al. [27].

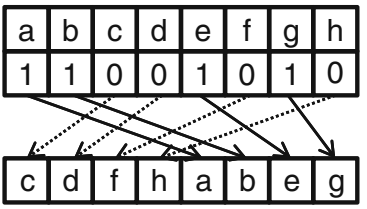

Fig. 9.4 An 8-bit group operation from Shi et al. [51].

- `GRP rd, rs, rc`
  permutes its data input `rs` into its data output `rd`, by grouping to the right those bits flagged with a 1 in a configuration input `rc`, and grouping to the left those bits flagged with a 0 (see Figure 9.4). A series of $log(n)$ `GRP` instructions can generate any permutation.

The `CROSS` [59] and `OMFLIP` [58] instructions use a virtual Beneš network or omega-flip network, respectively, to permute the n data bits. A n-stage Baneš network for permuting n bits is a butterfly network followed by an inverse butterfly network, each of which has $log(n)$ stages (Figure 9.5). An omega-flip network is isomorphic to a Beneš network. Since each `CROSS` or `OMFLIP` instruction executes the equivalent of two stages of the network, both can achieve any of the $n!$ permutations in at most $log(n)$ instructions.

- `CROSS.m1.m2 rs, rc, rd`
  executes the operation of the two butterfly stages specified by m1 and m2 that are characterized by a "butterfly width" of 2^{m1} and 2^{m2} respectively. The bits to be permuted are in `rs` and the permuted bits are in `rd`, while `rc` holds two groups of configuration bits (the left $n/2$ configuration bits in `rc` are for $m1$ and the right $n/2$ configuration bits in `rc` are for $m2$. There are $n/2$ butterflies in each n-bit stage; each configuration bit in a group specifies if the butterfly will propagate data into the straight path (0) or into the cross path (1) of each butterfly. By chaining `CROSS` instructions in sequence, a Baneš network for a desired permutation can be configured.

- `OMFLIP.c rd, rs, rc`
  The 2-bit subopcode `c` indicates which two basic operations are used in this instruction. For each bit, 0 indicates that an omega operation is used and 1 indicates that a flip operation is used. There are four combinations of `c`: omega-omega, omega-flip, flip-omega and flip-flip. The first basic operation takes the source register `rs` and moves the bits in it based on the least significant half of the configuration register `rc` to an intermediate result. The second basic operation moves the bits in the intermediate result according to the most significant half of `rc` to the destination register `rd`.

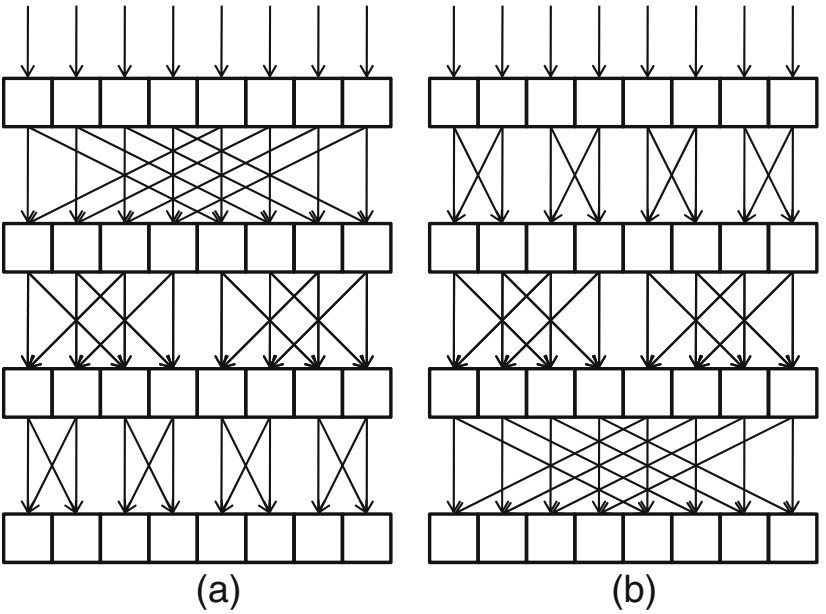

Fig. 9.5 (a) 8-input butterfly network; (b) 8-input inverse butterfly network from Shi et al. [52].

When comparing `OMFLIP` instructions versus table lookup, Yang et al. [58] achieved a 1.33 speedup for DES encryption/decryption and 16.55 speedup on DES key scheduling for a 2-way superscalar architecture with 1 load-store unit and a cache system similar to Pentium III processor. An omega-flip network can achieve the same performance as a `CROSS` instruction with a far smaller hardware implementation, according to [39].

Other proposed instructions are `PPERM` [39], `SWPERM` with `SIEVE`, `BFLY`, and `IBFLY`. Some fast implementations of these permutation instructions (including `CROSS`, `OMFLIP`, `GRP`) is proposed in the work of Hilewitz et al. [28]. The `GRP` can also be used to perform hardware radix sorting and has strong inherent differential cryptographic properties, but the implementation is relatively slow. On the other hand, the `BFLY/IBFLY` have a relatively fast implementation, as they can perform their operation in a single cycle.

The `PPERM` instruction [39] represents an intuitive way to do permutations (Figure 9.6), where the position of each bit in the destination `rd` is specified by a bit configuration register `rc`, `rs` being the source. This is similar to the `PERMUTE` instruction in the MAX-2 ISA extension [38]:

- `PPERM.x rs, rc, rd`
  permutes its input `rs` according to an explicit list of indices that are packed (in k groups) into `rc`; $x = 0, 1, \ldots, 7$ is a byte offset that specifies which k contiguous bits in `rd` will receive the source bits given by `rs`, while the remaining bits in

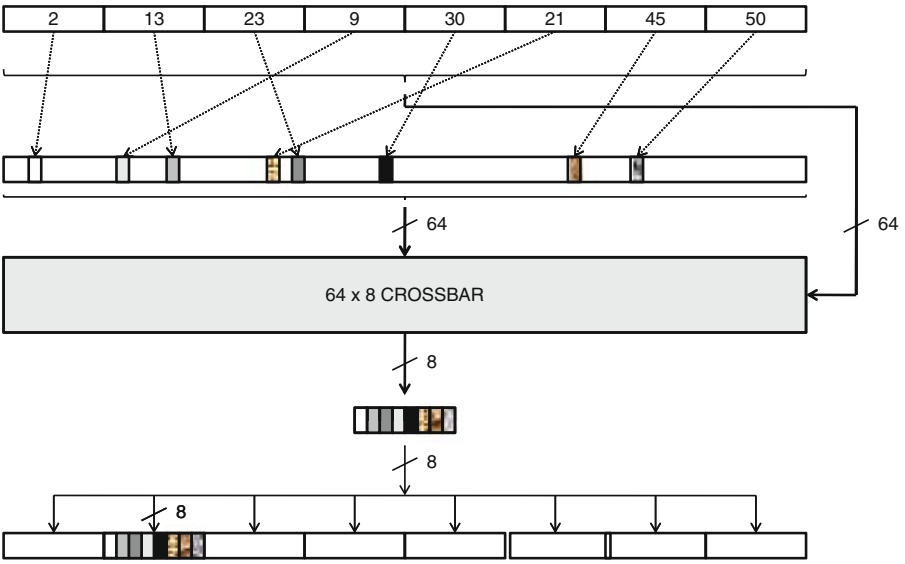

Fig. 9.6 Diagram flow of bits for `PPERM.1 R1,R2,R3`; the number in `R2` are the bit positions in `R1` [39].

`rd` are zeroed. Each index in the list `rc` specifies where in `rs` to extract each of the k bits (being $k <= n/log(n)$ for n-bit information, at most $n/log(n)$ indices are available in each instruction). Therefore, to specify a generic permutation, at most log(n) instructions are needed.

In a similar way the `SWPERM` with `SIEVE` [42] selects a source bit by its numeric index.

In the work of Shi et al. [52], the `BFLY/IBFLY` instructions are proposed. These can be useful for example in P-box permutation of DES cipher round function. By cascading these two instructions, a generic n-bit permutation can be achieved. While the implementation is fast, an ISE issue could be how to supply the configuration bits for the butterfly network. For $n = 64$, a 6-stage butterfly network requires $3n$ configuration bits, that is $n/2$ bits for each stage.

- `BFLY rd, rs, ar.b1, ar.b2, ar.b3`
- `IBFLY rd, rs, ar.b1, ar.b2, ar.b3`
  permute their input `rs` (and leave the results in `rd`)using a butterfly and inverse butterfly circuit, respectively (Figure 9.5). Assuming a 6-stage network, three extra special registers are needed (`ar.b1, ar.b2, ar.b3`, which contain the configuration bits). The configuration bits have similar meaning as in the `CROSS` instruction discussed above.

9.3.2 ISE for AES

Private-key (symmetric-key) cryptosystems are typically devoted to the encryption/decryption of the bulk of communications between two parties, while public-key cryptosystems are usually employed for authentication, exchange of the session key of symmetric algorithms, digital signature and other security operations involving a limited amount of information. One of the reasons for this is that public-key approaches are far more computationally expensive than private-key counterparts. However, as the bulk data encryption/decryption is performed by private-key algorithms, it is very important to implement these techniques very efficiently.

Recent research proposals analyzed ISEs, as well as specific coprocessors for speeding up these cryptosystems. AES [9] and the older DES [13] and triple-DES, or TDEA [14], block ciphers are and have been widely used for symmetric-key cryptography. The Rijndael block cipher [9] was accepted in 2000 as the new advanced encryption standard (AES) by the U.S. government and standardized by FIPS in 2001 [15]. AES is adopted in an increasing quote of applications and systems because of its security properties, flexibility and good implementability features on a wide range of architectures (*e.g.*, from 8-bit processors and up) both in hardware and in software. For these reasons we will focus mainly on AES in this section.

AES is a block cipher that operates on 128-bit blocks and can support 128, 192 and 256-bit keys. The 128-bit block is seen as a 4x4 matrix of bytes (*state*) and the encryption/decryption operations work on such a matrix, performing permutations on the rows, on the columns, and substituting bytes in it according to modular arithmetic with polynomials. The particular set of elaborations is briefly summarized in Figure 9.7 as pseudocode. Note that some operations (*rounds*) are performed a number of times (*e.g.*, 10 in the case of 128-bit key).

Initially, the *state* is the input block and then, it represents the intermediate results of the rounds. At the end, the *state* represents the encrypted block.

The SubBytes operation updates each byte in the array using an 8-bit S-Box, which introduces the non-linearity in the algorithm. The S-Box is derived from the multiplicative inverse over GF(2^8) and can be calculated directly or through a look-up table.

The ShiftRows operation rotates the rows of the state cyclically to the left. The first row is not shifted, the second row is shifted by 1 byte, the third by 2 bytes and the last row by 3 bytes.

In the MixCol operation, the four bytes of each column of the state are combined as follows. Each element of a column is considered as a polynomial over GF(2^8) and multiplied modulo x^4+1 with a fixed polynomial $c(x) = 3x^3+x^2+x+2$.

The AddRoundKey operation mixes a key, which is derived at each round from the main key, to the state.

The encryption and decryption steps are very similar and are based on the same set of base operations. Full details on AES internals can be found in the reference literature [9, 15].

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

```
AES_encrypt(in, out, w)
byte in[4*4],
byte out[4*4],
word w[4*Nr+1]
{
    byte state[4,4];
    state = in;

    // Initial AddRoundKey
    AddRoundKey( state, w[0, 3] );

    for round = 1 step 1 to Nr-1  // (Nr-1) rounds
    {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, w[round*4, (round+1)*3]);
    }

    // Last round (no MixColumns)
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, w[Nr*4, (Nr+1)*3]);

    out = state;
}
```

Fig. 9.7 Implementation of the inner loop of schoolbook multiplication in GF(p).

The AES was thought of for easy implementation and execution efficiency on both 32-bit microprocessors, as well as on small 8-bit microcontrollers. However, each round of the algorithm requires a significant number of assembler instructions because of the specific features of the involved operations. This fact limits the encryption/decryption bandwidth achievable in software. Highly optimized software approaches tend to employ look-up tables, which are acceptable for general-purpose systems but can be inadequate for memory-constrained embedded devices. However, when high speed is required, software implementations are too inefficient and ad hoc hardware approaches (*e.g.*, cryptoprocessors) are preferable. However, a crypto-processor might lack the flexibility to accommodate to different algorithm parameters (*e.g.*, key size) and, typically, is not able to run general-purpose code.

Using specifically extended instruction-sets and functional unit extensions on existing processors, significant performance gains are achievable and complete compatibility with the unmodified general-purpose processor can be maintained, keeping low the design complexity.

9.3.2.1 Extended Instructions

Bertoni et al. in [2] propose a word-level ISE for an ARM processor. Given its generality, the approach could be applicable to almost all 32-bit processors. As a baseline, they focus on a software implementation which relies on three look-up tables: two for the values of the nonlinear transformation and its inverse, and one for storing constants for the key schedule. Based on the execution time profiling of this software version, the most time-consuming code parts have been moved into hardware and specific instructions have been added to trigger the new hardware units.

They propose to include a couple of instructions: SBox and SMix which perform S-Box and both S-Box and MixColumns transformations in a single step, respectively. First, they propose a byte-oriented approach, where the new hardware and the new instructions work on one byte at a time. Then, they extend their proposal to 32-bit words, modifying both the hardware and the instructions and exploiting in this way the parallelism of the architecture and of the algorithm.

The byte-oriented instructions operate on only one byte of the AES status at a time, as shown in Figure 9.8. As the processor registers are word-oriented, a *Selector* block selects the right byte from the register word. After the byte is extracted, the nonlinear transformation (bytewise S-Box) is applied.

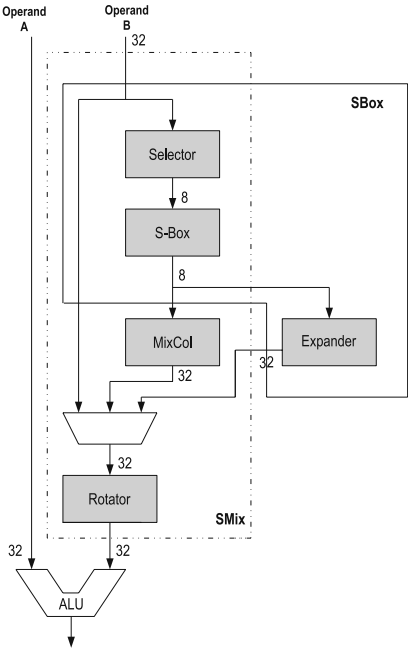

Fig. 9.8 Circuit structure to support *SMix* and *SBox* instructions.

The MixCol module processes the byte and completes the SMix instruction: it outputs four bytes, representing the different contributes. At every step of the AES, the polynomial changes but the multiplicative coefficients remain the same. Therefore, the correct result can be obtained through a reordering step of the bytes. The *Expander* on the right in Figure 9.8 is responsible to expand the processed byte to a word. If X is the input byte, 000X is the output word of the expander. In this way, its output can be sent to the final *Rotator* which is able to position the nonzero byte in the needed position of the result through an XOR. In this way, the four bytes of an output word (*i.e.*, 000A, 00B0, 0C00, D000) can be produced independently and XOR-ed together to form the output word (*i.e.*, DCBA). The suggested byte instructions are:

- **SBox Rs, Rd, Index**, which performs the nonlinear substitution on a byte. Four of them are needed to process an entire 32-bit word. *Rs* indicates the source register, *Rd* represents the destination register, and *Index* indicates the byte to extract and configures the *Rotator* operation. First, the accumulator must be initialized to "0000" and a load instruction is needed to initialize the source register. This instruction is used only in the key-scheduling phase and in the last round, where the MixColumns transformation must be skipped.
- **SMix Rs, Rd, Index**, which performs both the Sbox and the MixColumns transformations on the selected byte. As in the previous case, *Rs* indicates the source register , *Rd* represents the destination register and *Index* selects the correct byte. Here, the *Expander* is not needed since the output from the MixColumns module is 32-bit wide. This module can be used to produce all the needed contributes since the rotator is responsible for their reordering.

The assembler code to perform an AES round using the presented instruction extensions is shown in Figure 9.9.

Moving to a word-oriented approach, the MixColumns transformation can be easily extended for producing the new columns, while the ShiftRows transformation is not straightforward since it works in the orthogonal direction of the state matrix than the MixColumns one. The authors propose to solve the problem by modifying four fixed registers of the CPU in order to allow access to a single byte in each of them in parallel, thus actually retrieving the word exactly as if it was passed through the ShiftRows transformation. From the implementation point of view, this implies modifying the register file of the processor and can be more or less difficult, depending on the internal architecture of the processor.

For the SubWord (S-Box) transformation, a word-level S-Box module is used. As in the byte-oriented solution, the MixCol module is cascaded to it. The rotator is still needed for the key-scheduling process. In this step, a word of the key contribute must be rotated and processed by the S-Box module. The final result is obtained through an *XOR* operation in the ALU.

The three word-level instructions needed are:

- **SMixW N, Rd**, which performs the S-Box and the MixColumns transformation in sequence. It does not need source registers as operand, since the registers containing the state are fixed due to the changes performed in the register file to

```
# Round computation                    # Key unroll
 1: SMix R0, R0, 0   ▷ 1st column      1: SBox R10, R4, 3   ▷ word rot. + S-box
 2: SMix R0, R0, 1                     2: SBox R10, R4, 0
 3: SMix R0, R0, 2                     3: SBox R10, R4, 1
 4: SMix R0, R0, 3                     4: SBox R10, R4, 2
 5: Xor R0, R4                         5: XOR R10, R8       ▷ Add round const
 6: SMix R1, R1, 0   ▷ 2nd column      6: XOR R4, R10       ▷ Calculate words of next
 7: SMix R1, R1, 1                        contribute
 8: SMix R1, R1, 2                     7: XOR R5, R4
 9: SMix R1, R1, 3                     8: XOR R6, R5
10: XOR R1, R5                         9: XOR R7, R6
11: SMix R2, R2, 0   ▷ 3rd column
12: SMix R2, R2, 1
13: SMix R2, R2, 2
14: SMix R2, R2, 3
15: XOR R2, R6
16: SMix R3, R3, 0   ▷ 4th column
17: SMix R3, R3, 1
18: SMix R3, R3, 2
19: SMix R3, R3, 3
20: XOR R3, R7
```

Fig. 9.9 Implementation of the inner loop of schoolbook multiplication in GF(p).

allow the selection. The arguments are the column index N and the destination register *Rd*.

- **SubWord Rs**, which performs only the S-Box transformation of the incoming word. It is used in the last round and in the key scheduling.
- **KSFW Rs, RCon**, which causes the word contained in the register *Rs* to feed the SubWord module; after that, it is rotated by one position and summed with the *RCon* constant stored in *RCon*, producing a word of the new key contribute. This represents the transformation of the first word of the key contribute.

The assembler code resulting from availability of these word-level instructions is shown in Figure 9.10.

Using the proposed ISE, the performance speedup in AES encryption is about 2.54x for AES-128. In particular, 497 cycles are needed using all the extended word-level instructions against 727 cycles when using the byte-level ones and 1771 cycles for the original software version. The speedup is very interesting, especially because it is obtained on top of a high-performance software implementation employing various look-up tables.

```
# Round computation        # Key unroll
1: SMixW 0, R10, R4        1: KSFW R4, R8
2: SMixW 1, R11, R5        2: XOR R5, R4
3: SMixW 2, R12, R6        3: XOR R6, R5
4: SMixW 3, R13, R7        4: XOR R7, R6
```

Fig. 9.10 Implementation of the inner loop of schoolbook multiplication in GF(p).

Another approach is proposed by Elbirt [12], who presents an ISE for Galois Field (GF) constant multiplication within a 32-bit SPARC v8 compatible processor. The approach can be applied to all the algorithms that use GF constant multiplication, not only to the AES. As the a central computation of AES rounds is the modular multiplication of GF(2^8) elements, the core of the proposal is to adopt an 8x8 *matrix* circuit that operates on the 8 bits of the input polynomial $a(x)$ and generate the 8 bits of the multiplication of $a(x)$ for the constant polynomial $k(x)$. The circuit implements a vector–matrix multiplication, where the 8 bits of $a(x)$ are the vector and the 64 coefficients of the matrix comprise both the bits of the constant $k(x)$ and the operations for executing also the modular reduction step. The total number of gates for such a circuit is about 7200, which is shown to be 18 times smaller than a look-up table approach.

The cycle count needed to perform an 8-bit modular multiplication comprising reduction is 1 cycle for the matrix approach, while, for instance, it is 28 cycles in software for a Pentium-class Intel processor. The authors show that the speedup over the software version without extension is 1.67x, and further improvements (up to more than 8x) are achievable if other extensions proposed in literature, like *S-Box*, are used together with the *matrix* extension.

Fiskiran and Lee in [17] analyze a variety of cryptographic algorithms and techniques for mobile devices. They consider symmetric-key, hash, public-key, ECC, and DSAs and analyze the main features of their software implementations in order to single out possible extensions to accelerate their execution. For the considered symmetric-key algorithms (AES, DES/3DES, RC4, Blowfish, MARS, Twofish), the workload characterization gives the following indications. Six of the 10 ciphers use table lookups, and 5 of these spend the largest fraction of their execution time during these table lookups (*e.g.*, 72% for AES, 58% for RC4). For all ciphers, tables are small and constant in size. The number of entries per table is 256 for 5 of the ciphers (8 index bits), and the data read is either 8 or 32 bits. Apart from RC4, tables are accessed only for reading. Moreover, the typical round structure of the ciphers allows also parallel table lookups. For example, all lookups in an AES round (16 lookups) can be performed in parallel, given enough hardware resources.

Based on this analysis, the authors propose an ISE for supporting table lookups in symmetric-key ciphers. The reference platform used is a PAX architecture, a minimalist high-performance cryptographic processor ISA designed at Princeton University [49], which features an ALU, a shift unit, a multiplier, and 8 on-chip tables (T0–T7). Each table has 256 entries and the size of each entry is equal to the processor word size, which may be 32, 64 or 128 bits.

The proposed lookup instruction for 32-bit word size is able to perform up to 4 simultaneous lookups and is named *ptlu* which is short for parallel table lookup. The exact format of the instruction is *ptlu.subword.table.offset.step Rd, Rs*, where the *subword* field selects the number of bytes to read from the table (size of data to read). The *table* field is 3 bits wide and selects the desired table. The byte-sized indices used to access the tables are read from the source register *Rs*, which is 32-bit

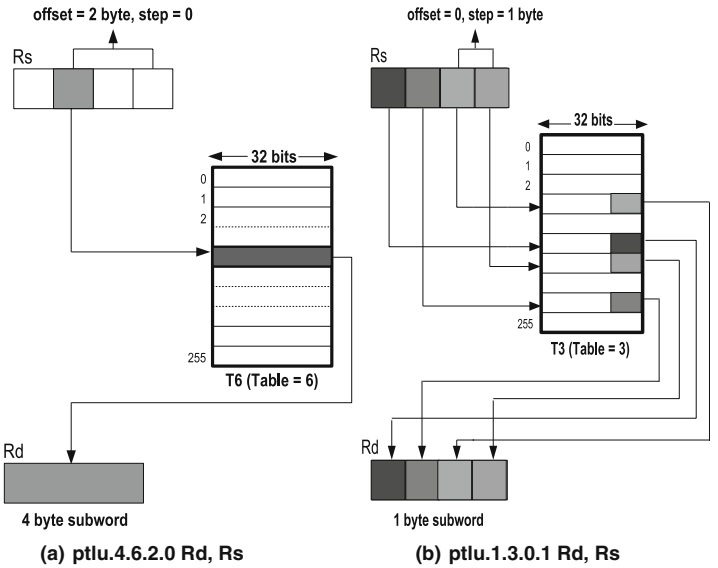

Fig. 9.11 A couple of examples of the *ptlu* instruction for accelerating table lookups.

long and so it can contain up to four byte-sized indices. The 4-bit *offset* field is used to select the first byte-sized index in *Rs*. The 4-bit *step* field gives the distance (in bytes) between two consecutive bytes in *Rs* that are used as indices when multiple lookups are performed.

For instance, in Figure 9.11 a couple of examples are shown. In 9.11a, a 4-byte access is shown using the byte 2 of *Rs* as index of T6, while in 9.11b, four parallel lookups in table T1 are made to read 1 byte per lookup and using the four bytes of *Rs* as indices. Note that the AES key expansion can take full advantage from this instruction to perform the byte substitution procedure.

The *ptlu* instruction is assumed to have 1 cycle latency, as it accesses a small on-chip memory. Using *ptlu*, the speedup of AES is 2.29x. Table 9.1 also shows that the other considered algorithms benefit from *ptlu* and deliver a significant speedup.

Table 9.1 Algorithm speedup using ptlu.

Algorithm	% Speedup
AES	2.29x
DES	1.28x
3DES	1.25x
RC4	1.92x
Blowfish	1.73x
MARS	1.40x
Twofish	1.61x

The authors highlight a possible advantage of *ptlu* instruction: its scalability to processors with different word sizes. In fact, as word size increases, more lookups can be performed simultaneously. For instance, if the word size is 128-bit, only 4 *ptlu* are needed to complete the 16 lookups of a single AES round. In case of 64 and 128-bit word sizes, the speedup for AES reaches 2.85x and 6.10x, respectively.

Tillich *et al.* in [55] analyze possible ISEs for AES on a 32-bit processor. Similar to [2], they propose byte-oriented and word-oriented instructions but with some differences. They propose byte-level *Sbox* and *MixCol* instructions, which calculate only one byte of the result. A specific immediate value of the instructions allows one to choose the source byte (*Sbox*), the destination byte and the encryption/decryption operation (for both *Sbox* and *MixCol*).

The *Sbox4* and *MixCol4* instructions extend this approach to 32-bit words and aim to parallelize four byte-operations. As shown in Figure 9.12, the *Sbox4* instruction substitutes all four bytes of the first source register and places them into the destination register. An optional byte-wise rotation can be performed on the result. The immediate value selects whether S-Box or inverse S-Box are used and specifies the rotation distance for the result. In this way, the key expansion step can be efficiently supported through this *sbox4* instruction. The *mixcol4* instruction calculates all four result bytes of the MixColumns or InvMixColumns operations, according to the immediate value.

The performance of the word-oriented ISE is better than the byte-oriented one but not as much as it could be expected due to the four-fold parallelism

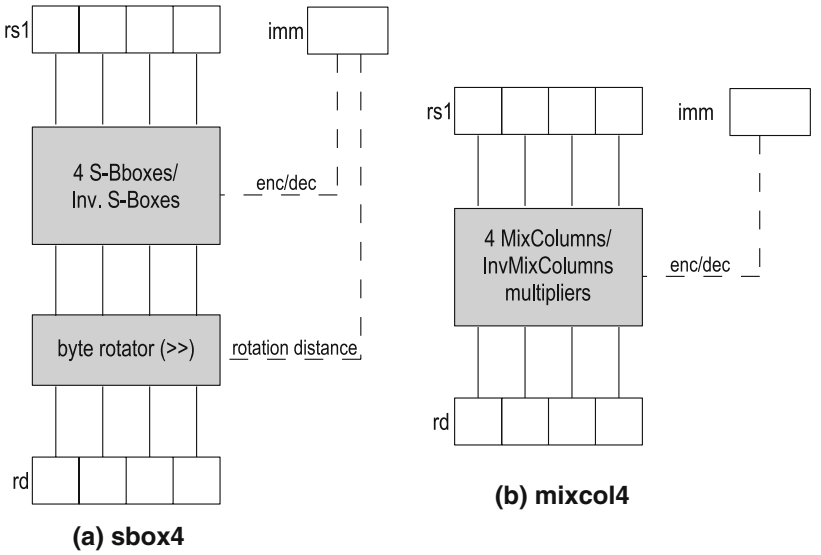

Fig. 9.12 Word-level *sbox* and *mixcol* instructions.

increase: 4.86x speedup vs. 1.74x speedup over the pure software implementation (no ISE). This is because the *Shift-Rows* AES operation becomes a bottleneck. Accelerating the most time-consuming parts of the algorithm caused other parts of the algorithm to become more important from the performance point of view.

The authors address this issue and propose a slight modification to the word-level circuits and instructions, so that they support a pair of input registers instead of only one. In this way, neglecting the details, the *Shift-Rows* transformation can be implicitly performed extracting 2 bytes from each of the two source registers. The speedup of this slightly improved ISE reaches 7.47x (8.35x with loop unrolling) over the non-extended software one and this is in line with expectations.

This experience should teach one that the ISE design and tuning is an iterative process that often has to be fed back from its own effects in order to obtain optimal results. The proposed approach does not rely on look-up tables and, therefore can be particularly suitable for constrained devices. The authors show that their ISE causes an 81% reduction in code size, while the well-known table-lookup approaches increase the code size by about 350–400%.

Tillich *et al.* in [54], explore the possibility of employing the ISEs for elliptic curve cryptography (ECC), assumed available on a given processor, to speed up AES. The considered ISE are:

- $gf2mul\ A,B$: 32-bit word-level polynomial multiplication, which generates a 63-bit result ($A \otimes B$) in the two 32-bit parts of an accumulator register $ACC.hi/ACC.lo$.
- $gf2mac\ A,B$: 32-bit word-level polynomial multiply and accumulate, which performs ($A \otimes B$), sums the result to $ACC.hi/ACC.lo$ and accumulates the outcome back to $ACC.hi/ACC.lo$.
- $shacr\ A$: shifts right the accumulator by 32-bit, so that $ACC.lo$ goes in A register and $ACC.hi$ goes in $ACC.lo$.

The MixColumns AES step requires to multiply two polynomials of degree 3 over GF(2^8), modulo x^4+1, where one polynomial is constant (specified by the algorithm) and the other is a column of the state. Essentially, each of such polynomials can be represented on a 4-byte word and the multiplication can be accelerated using $gf2mul$ instruction. After that operation, two more steps are needed: reduction of the polynomial coefficients, which live in GF(2^8), and polynomial reduction according to the modulus to obtain the correct 32-bit result.

The first operation can take advantage of $gf2mac$, while the second can benefit from *shacr*.

Considering that these extensions were not aimed specifically to AES, the performance increase from their usage is significant: +23% and +20% in the case of precomputed key schedule, for encryption and decryption, respectively.

The reader is highly encouraged to go into the details of the cited paper, as a useful exercise to fully understand the usage of the proposed approach.

9.3.3 ISE for ECC applications

Elliptic curve cryptography (ECC) was proposed independently by Victor Miller [44] and Neal Koblitz [33] in 1985 and is gaining interest as a viable alternative to "standard" public-key methods (like RSA), because of its shorter keys at the same security level. This can translate into faster implementations and reduced consumption of energy and bandwidth, which are crucial points, especially for embedded applications on constrained devices.

An elliptic curve for cryptography can be defined over a finite field (FF), *e.g.*, GF(p), GF(2^m), and is the set of points $P = (x,y)$ that satisfy the equation $y^2 + xy = x^3 + ax^2 + b$, $a,b,x,y \in FF, b \neq 0$, together with a point at infinity O. An addition operation defined on the curve points allows calculating integer multiples of a point: given a point P and an integer k, [k]P (*i.e.*, the scalar multiplication operation) produces another point Q on the same curve. Scalar multiplication can be naturally implemented through repeated doublings and additions of the point P, and it has security features similar to exponentiation in discrete-logarithm cryptosystems.

Elliptic curve point addition and doubling, in turn, can be calculated with a number of additions, multiplications, squarings, and inversions in the underlying binary finite field, through formulas operating on the coordinates of the involved points. For example, Figure 9.13 shows the formulas [43], needed to calculate EC addition and doubling in affine coordinates.

Handling finite-field elements in software requires multiprecision arithmetic because typical field sizes are hundreds of bits long. For instance, the sizes of EC binary fields delivering a security level similar to 1024, 2048 and 3076-bit RSA are 163, 233 and 283-bit, respectively.

Standard processors manage such big values (m-bit) as arrays of w-bit long words (where w typically is 8, 16, 32 or 64 bits, matching the word size of the processor). The number of required words is $\lceil m/w \rceil$.

Finite-field addition can be performed word by word, taking into account the carry propagation from each word to the immediately more significant one, in the case of GF(p). This can be tricky using a high-level language because there is no direct control over the carry propagation. In an assembler, according to the available ISA, it might be easier to obtain an efficient implementation. For instance, on Intel x86 processors, *ADC* instruction sums two operands plus the carry flag set by

$$
\begin{aligned}
x_3 &= \lambda^2 + \lambda + a + x_1 + x_2 \\
y_3 &= \lambda \cdot (x_1 + x_3) + x_3 + y_1 \\
\lambda &= \begin{cases} (y_2 + y_1)/(x_2 + x_1), & if P \neq Q \\ x_1 + y_1/x_1, & if P = Q \end{cases}
\end{aligned}
$$

Fig. 9.13 Elliptic curve point addition and doubling formulas, given $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $P + Q = (x_3, y_3)$.

the previous instruction, allowing straightforward implementation of multiprecision additions. In case the result of an addition is bigger than the field prime, a reduction operation must be performed.

In GF(2^m), addition is simpler because it becomes a bitwise exclusive-OR and, therefore, it does not require to manage carries.

Field multiplication is quite expensive because, by nature, it *mixes* all the words of the two operands together and outputs a double-sized value. Multiplication can be done iteratively via repeated shifts and additions/XOR (*i.e.*, one per bit) to the partial product, as in the well-known schoolbook method for base 10 numbers. In each iteration, a number of instructions are needed to process the intermediate field values.

In the case of GF(p) fields, the algorithm can work at word level and be much more efficient. In fact, it can take advantage from the instructions and circuits available in the processor for integer multiplication (w-bit operands) so that the m-bit multiplication can be implemented via a number of w-bit multiplications. In the case of GF(2^m) this is not typically possible because there are almost no processors that natively support operations (*e.g.*, multiplication) on binary fields.

As a matter of fact, multiprecision modular multiplication, together with the modular inversion, may take most of the execution time of EC operations implemented in software [4, 25]. Therefore, their performance is crucial for the efficiency of high-level cryptographic protocols. Inversions are much more expensive computationally than multiplications but they can be almost totally avoided using a projective representation of the curve.

Concluding, ordinary processors are not particularly suited for efficient working on multiprecision operands and can be very inefficient in managing binary field GF(2^m) operations. For these reasons, specific ISEs have been studied to improve the ECC performance on these platforms.

9.3.3.1 Extended Instructions

Fiskiran and Lee in [16] do an extensive study on the performance of the elliptic curve public-key cryptosystem and, in particular, on the performance of the binary field GF(2^m) operations needed by ECC. The elements of a binary extension field are polynomials with coefficients from $\{0, 1\}$, having high degree (*e.g.*, 162, 192, 232). They highlight that the main bottleneck of performance is due to the fact that key arithmetic operations on these polynomials, such as squaring and multiplication, are not supported by integer-oriented processor architectures. Instead, these are implemented in software, causing a very large fraction of the cryptography execution time to be dominated by a few elementary operations. For example, more than 90% of the execution time of 163-bit ECC may be consumed by two simple field operations: squaring and multiplication. The authors derived these results from an implementation of ECC on an Intel Pentium-II workstation. They used the Montgomery scalar multiplication algorithm described by Lopez *et al.* in [40] and which is the fastest method that does not require significant precomputations and/or storage.

They show that typical high-level application benchmarks for security protocols (EC Diffie-Hellman key-exchange protocol, EC digital signature generation/verification, and El-Gamal encryption/decryption) spend from about 94% up to 99% of the time in EC scalar multiplication k[P]. For this reason, the performance of this EC operation can be addressed in isolation as a good measure for overall application performance.

Using an instrumented version, through *gprof* GNU tool, of the scalar multiplication benchmark, the authors were able to break down the execution time of k[P] on the Pentium-II into the time components spent in the various code regions. In this way, it was possible to measure the time spent into squaring (with reduction), multiplication (with reduction), inversion and other modular operations. Table 9.2 shows these results and highlights that modular squaring and multiplication are the most important as they take 6.33% and 87.25% of the total execution time, respectively. Therefore, according to Amdahl's law, modular multiplication is the first target of possible optimizations for speeding up ECC.

The authors first analyze the intrinsic features of these field operations in order to understand the limits achievable through more complex processor microarchitectures. Using SimpleScalar [5], a cycle-accurate simulator for computer architecture, they simulate single-issue and multiple-issue processors (*i.e.*, capable of managing more than one instruction per cycle due to the parallelism in their pipeline). Results show that the critical operations (multiplication and squaring) implemented in software can have a speedup of more than 1.9x for a two-way execution core and more than 3.4x for a four-way one. This indicates the presence of a good number of independent instructions close to each other in the dynamic instruction flow of these benchmarks. The independent instructions are likely to originate from the operations on the various words that make up the field elements (*e.g.*, six 32-bit words for each 163-bit field element).

In addition, the authors show that adding one more load/store unit, to increase the memory bandwidth, brings almost no benefits to a two-way issue machine, while it improves slightly the performance of a four-way issue machine. Obviously, these results are very biased by the particular software implementation of the field operations. For instance, software implementations that rely on look-up tables of precomputed results are likely to benefit more from memory bandwidth increase. However, these results show that a relatively complex general-purpose processor can improve ECC performance significantly compared to a simple one.

Then, the authors investigate the effects of including a specific ISA support for word-level polynomial multiplication. They propose a *bfmul* instruction that works

Table 9.2 Execution time quote of finite-field operations in EC scalar multiplication $k[P]$.

Operation	% of Total Execution Time
Squaring (including reduction)	6.33
Multiplication (including reduction)	87.25
Inversion	1.51
Other	4.91

on two 32-bit operands and outputs the 64-bit product on a couple of special 32-bit registers *RH* and *RL*. The speedup of polynomial multiplication using this instruction on a single issue processor is almost 25x. The authors also analyze a simpler unit that can write only 32-bits of the result at a time. In this case, even if two instructions *bfmul.lo* and *bfmul.hi* are needed to obtain the lower and higher word of the result, respectively, the achieved speedup is still more than 24x. This indicates that even a narrower interface between the multiplier unit and the existing datapath is able to be effective.

In addition, the authors highlight that, using a *rev* instruction, which reverses the bit order within a word (*i.e.*, the most significant bit becomes the least significant one and so on), it is possible to substitute the *bfmul.hi* instruction by three *rev* instructions (two of them can be executed in parallel), a *bfmul.lo* and a shift instruction. In this way, the circuit can be further simplified because the circuit for bit reversing is far simpler than the one for *bfmul.hi*, as it does not need any logic gate but only wiring. Note that, for a number of DSP processors, the *rev* instruction comes for free because it is already part of their standard ISA. This is the code fragment that can substitute the *bfmul.hi* instruction:

```
rev t1, a       # t1, t2 are temporary variables
rev t2, b       # two rev instructions are independent
bfmul.lo t1, t1, t2
slli t1, t1, 1  # 1-bit logical shift left
rev t, t1
```

This approach allows a speedup of more than 17x over the single-issue machine, which is significantly lower than in the case of the other more complex approaches, but it is still a very good solution for designs where the hardware modifications have to be kept as small as possible.

In the same paper, the authors also address a possible extension for the the squaring operation in binary fields. Recall that the square of a binary polynomial has the same bits as the operand, but with zeroes interleaved between each pair of them. This can be easily accomplished by the *shuffle* instruction inspired by the DSP world. Such instruction reads one bit at a time alternatively from the two source registers and inserts them into the destination register. In this way, if the first register holds the polynomial coefficients and the second one holds zero, the output result is the squaring of the polynomial.

Again, some issues might come into the game because of the width of the output. In order to maintain the *shuffle* instruction with one-word output, it must work on half-word operands. For instance, *shuffle.lo* and *shuffle.hi* instructions could be provided so that the low and the high half-words could be processed, separately. Actually, the *shuffle.hi* can be emulated with a pair of preliminary shifts on the operands and using the *shuffle.lo*, reduce the overall circuit complexity.

The speedup of finite field squaring using the *shuffle* instruction is 3.8x over the software implementation.

Fiskiran and Lee further investigate the usage of complex processor microarchitectures when the above-mentioned extensions are available. A two-way issue processor, with either one or two load/store units and one or two multiplier units,

deliver about 1.75x speedup, which is slightly lower, but still in line with the results of the unmodified processor. In comparison, a four-way issue processor, with one or two load/store units and one or two multipliers, gives a speedup of about 2.2x, which is significantly lower than in the case of the software version on the unmodified processor. This means that the software that uses the ISE has a lower instruction-level parallelism (ILP) than the original one and, because of that, complex wide-issue general-purpose architectures are less beneficial in this case. However, the adoption of *bfmul* and *shuffle* instructions allow one to speed up the application-level benchmark (EC point multiplication) by more than 7.5x for a single issue processor and up to more than 22x for a four-issue processor with four ALUs, two load/store units, and two multiplier units. The adoption of the simpler variants of *bfmul* and *shuffle* do not affect these results too much.

The possible benefits from the usage of a specific optimized operation for word-level polynomial multiplication were first highlighted by Koç *et al.* in [34], where this operation was named MULGF.

Großshädl et al. in [22] highlight a proposal for fast ECC over binary fields GF(2^m) on a possible 16-bit smart card architecture. The contribution highlights the difficulty of implementing polynomial multiplication on the limited general-purpose architecture of a smart card. The authors propose the integration of a word-level polynomial multiplier unit within the existing datapath of the processor. They highlight that the overhead in hardware complexity can be very limited because the polynomial multiplier can share the same circuit as the integer one if a dual-field adder (DFA) [48] is used.

In fact, the DFA, shown in Figure 9.14 for only one bit addition, can be employed within the multiplier circuit so that the partial-product accumulation can be done using integer additions, when *fsel=1*, or polynomial additions (*i.e.*, XOR operations), when *fsel=0*. The authors highlight that the selection logic of a dual-field adder increases the area and the delay compared to a standard full adder, but when it works in polynomial mode, two NAND gates are forced to one and only the two XOR gates contribute to the dynamic power consumption. In this way, if the power consumption due to leakage is negligible compared to dynamic power, the DFA consumes significantly less power when working in polynomial mode than in integer mode.

Given the availability of the word-level multiplier for polynomials, and the corresponding new instruction *MULGF2*, word-level algorithms for multiprecision multiplication, squaring and modulo reduction can be employed in place of the bit-oriented ones. In this way, the machine parallelism can be fully exploited also in polynomial computations. For instance, as shown in Algorithm 1, the word-level schoolbook pencil-and-paper method for polynomial multiplication translates into a number of *MULGF2* instructions, marked as $\otimes$, and word-level *XOR* instructions, marked as $\oplus$. Any iteration of the inner loop carries out an operation of the form $(\tilde{u},\tilde{v}) \leftarrow \tilde{r}_{i+j} \oplus (\tilde{a}_j \otimes \tilde{b}_i) \oplus \tilde{u}$. The tuple $(\tilde{u},\tilde{v})$ is a double-precision quantity representing $u(t) \cdot t^w + v(t)$ (*i.e.*, a polynomial of degree 2w-1, where w is the processor word width).

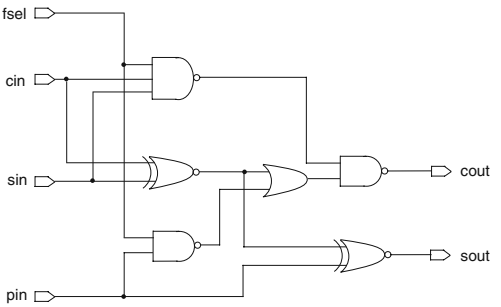

Fig. 9.14 Structure of a dual-field adder.

Algorithm 1 Pencil-and-paper method.

Input: Binary polynomials $a(t) = (\tilde{a}_{s-1}, ..., \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, ..., \tilde{b}_0)$ consisting of s word each
Output: Product $r(t) = a(t) \otimes b(t) = (\tilde{r}_{2s-1}, ..., \tilde{r}_0)$.

1: $r(t) \Leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: $\quad \tilde{u} \Leftarrow 0$
4: $\quad$ **for** j from 0 by 1 to $s-1$ **do**
5: $\quad\quad (\tilde{u}, \tilde{v}) \Leftarrow \tilde{r}_{i+j} \oplus (\tilde{a}_j \otimes \tilde{b}_i) \oplus \tilde{u}$
6: $\quad\quad \tilde{r}_{i+j} \Leftarrow \tilde{v}$
7: $\quad$ **end for**
8: $\quad \tilde{r}_{s+i} \Leftarrow \tilde{u}$
9: **end for**
10: **return** $r(t)$

The Comba multiplication method [7] is shown in Algorithm 2. Comba's method forms the product $r(t)$ by computing each word $\tilde{r}_i$ of the result at a time, starting with the least significant word $\tilde{r}_0$. The partial products $\tilde{a}_j \oplus \tilde{b}_i$ are processed by columns instead of by rows as in the schoolbook pencil-and-paper method.

Comba's method for long integer multiplication can deliver performance advantages on processors having a multiply/accumulate unit (*e.g.*, digital signal processors) [18] because each word of the result is calculated by repeated multiply/accumulate (MAC) operations. Another potential advantage of Comba's method originates from keeping the running sum $(\tilde{u}, \tilde{v})$ in a register pair because, in this way, no store instructions are needed during the multiply/accumulates for computing each result word. Conversely, each iteration of the inner loop of schoolbook algorithm requires three memory accesses in order to load the values of $\tilde{a}_j$ and $\tilde{r}_{i+j}$, and write back the result to $\tilde{r}_{i+j}$.

In other words, Comba's method eliminates the write-back operation by changing the order of partial-product generation/accumulation such that each word of *r(t)* is computed completely before passing to the next one.

However, possible drawbacks of Comba's method can originate from the reversed addressing of the words of the operands *a(t)* and *b(t)*, along with the more complicated loop control. Nevertheless, on processors that support an

Algorithm 2 Comba's method for binary polynomials.

Input: Binary polynomials $a(t) = (\tilde{a}_{s-1}, ..., \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, ..., \tilde{b}_0)$ consisting of s word each
Output: Product $r(t) = a(t) \otimes b(t) = (\tilde{r}_{2s-1}, ..., \tilde{r}_0)$.

```
 1: (ũ, ṽ) ⇐ 0
 2: for i from 0 by 1 to s − 1 do
 3:     for j from 0 by 1 to i do
 4:         (ũ, ṽ) ⇐ (ũ, ṽ) ⊕ (ã_j ⊗ b̃_{i−j})
 5:     end for
 6:     r̃_i ⇐ ṽ
 7:     ṽ ⇐ ũ, ũ ⇐ 0
 8: end for
 9: for i from s by 1 to 2s − 2 do
10:     for j from i − s + 1 by 1 to s − 1 do
11:         (ũ, ṽ) ⇐ (ũ, ṽ) ⊕ (ã_j ⊗ b̃_{i−j})
12:     end for
13:     r̃_i ⇐ ṽ
14:     ṽ ⇐ ũ, ũ ⇐ 0
15: end for
16: r̃_{2s−1} ⇐ ṽ
17: return r(t)
```

auto-increment/decrement addressing mode, the first drawback might be canceled because the computation of the addresses of $\tilde{a}_j$ and $\tilde{b}_{i-j}$ comes for free.

The effects on performance of the memory behavior of Comba's method depend on the features and speed of the memory subsystem (*i.e.*, cache, if any, buses and RAM chips) of the target processor. It is very interesting to observe that the actual performance of different algorithms can be highly affected by the features of the execution platform: the availability of special instructions, as well as, the organization and speed of the memory system. Therefore, the selection, and the tuning, of the most performing algorithm must take into account very precisely the architecture of the system that will be running the specific implementation of the algorithm.

Großshädl et al. in [56] use ECC as a case study for analyzing when ISE can change algorithm design. The main results of the work are the following. First, ISE can reverse the relative interest of different algorithm versions. Second, automatic exploration of the best ISE can be viable for an algorithm designer in this field. In fact, they show that the considered automatic exploration tool was able to derive similar results as the ones obtained through manual exploration and simulation, even if with less accuracy in the performance estimates.

The authors present possible alternative algorithms for polynomial multiplications both in GF(p) and GF(2^m) finite fields and show how, in both cases, specific ISEs can help to boost performance.

For their experiments, the authors use the MIPS32 architecture, which has a multiply-and-accumulate integer instruction (*MADDU*) that can be very useful in the GF(p) multiplication algorithms. The *MADDU* instruction was aimed to DSP-like computations and operates as follows: multiplies two 32-bit words, adds the product to the 64-bit value in the concatenated *HI*/*LO* register pair, and writes back

```
L1: lw      $t0, 0($t1)     #load A[j]
    addiu   $t1, $t1, 4     #increment A pointer
    multu   $t0, $t4        #A[j]*B[i]
    lw      $t2, 0($t3)     #load Z[i+j]
    maddu   $t5, $t7        #add old U to product
    maddu   $t2, $t7        #add z[i+j] to product
    addiu   $t3, $t3, 4     #increment Z pointer
    mflo    $t6             #read V
    mfhi    $t5             #read U
    sw      $t6, -4(t3)     #write V to Z[i+j]
    bne     $t1, $t8, L1    #if j!= s branch to L1
```

Fig. 9.15 Implementation of the inner loop of schoolbook multiplication in GF(p).

the result to *HI*/*LO*. Using the *MADDU* instruction, the core-loop of the schoolbook pencil-and-paper method ($a \cdot b + z + u$, where a, b, z, u are 32-bit values) can be mapped on only 11 instructions as in Figure 9.15. The main residual problem in this method is that the two additions of 32-bit values to the 64-bit product might generate carry bits and so they must be managed as double-precision additions, with increased complexity.

The authors describe that if the instruction set could be augmented with an instruction that is able to perform multiply, accumulate and the two additions (named *MADDL* as in [20]), the inner loop could be made even faster. This extension can increase slightly the area and the delay of the circuit but could shrink the code shown in Figure 9.15 from 11 down to only 7 instructions. Apart from the details that can be found in [20], if load/stores are assumed to hit in the data cache in both cases, all instructions can be assumed to run in one cycle. In this way, *MADDL* instruction can enable a significant speedup (1.57x) over the code that uses *MADDU*.

For the inner loop of Comba's method on GF(p) fields, the *MADDU* instruction appears to provide exactly the functionality needed by the concatenated multiply/accumulate operations. However, multiple accumulations on the 64-bit HI/LO register pair cannot be done without possible overflow. For this reason, even using *MADDU* instruction, the inner loop of Comba's takes no less than 18 cycles. However, another simple custom instruction extension could be very beneficial in this case. If the accumulator of the *MADDU* (*HI*/*LO* pair) is widened, for instance *HI* register is extended to 40-bit, up to 256 *MADDU* could be performed without risk of overflow. This could be enough for ECC applications and would not increase significantly the complexity and the delay of the circuit. Großshädl et al. highlight that a careful implementation of the Comba core loop, employing this modification, could be run in only 6 clock cycles. In this way, the possibility of customizing the instruction set according to the algorithm features allows the slowest algorithm with unmodified ISA (Comba) to be the absolute fastest when a custom-tailored ISA is employed.

On GF(2^m) fields, the authors perform a similar analysis and show that, using the standard ISA, the inner loop of the polynomial multiplication can execute in only 10 cycles. It seems in line with GF(p), but the difference is huge because the inner

loop on GF(2^m) is performed many more times because of the lack of word-level instructions for polynomials. The number of iterations is a function of the number of bits of the operands instead of the number of words, as shown in Algorithm 3 for the shift-and-XOR multiplication algorithm.

Several improvements on this method have been proposed and, essentially, they rely on precomputed look-up tables which limit the number of operations. The look-up table must be calculated online because it contains multiples of $a(t)$ and, therefore, the table length has to be tuned taking into account the trade-off between the pre-computation overhead and the resulting performance benefits. This trade-off is certainly dependent on the particular processor features. For instance, in [26, 41] the best trade-off is shown to be a 16-entry table (*i.e.*, 16 multiples of $a(t)$), which allows it to process $b(t)$ four bits at a time. In this way, only 1/4 of the multiprecision shifts are needed, where each one is a four-bit shift.

The usage of a look-up table causes the memory traffic to increase in the inner loop and increases the overall memory footprint of the algorithm.

Algorithm 3 Pencil-and-paper method.

Input: Binary polynomials $a(t) = (\tilde{a}_{s-1}, ..., \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, ..., \tilde{b}_0)$ consisting of s words each
Output: Product $r(t) = a(t) \otimes b(t) = (\tilde{r}_{2s-1}, ..., \tilde{r}_0)$.

```
1: r(t) ⇐ 0
2: for i from 0 by 1 to s − 1 do
3:     ũ ⇐ 0
4:     for j from 0 by 1 to s − 1 do
5:         (ũ, ṽ) ⇐ r̃_{i+j} ⊕ (ã_j ⊗ b̃_i) ⊕ ũ
6:         r̃_{i+j} ⇐ ṽ
7:     end for
8:     r̃_{s+i} ⇐ ũ
9: end for
10: return r(t)
```

If a word-level approach is also used in GF(2^m), the performance of the multiplication algorithm becomes strictly related to the efficiency of the MULGF2 operation (*i.e.*, multiplication of two 32-bit polynomials, obtaining a 63-bit result), which must be implemented in software using the standard processor ISA. The authors of [56], highlight that the sole MULGF2 operation can be emulated in no less than 190 cycles on their MIPS32 processor, in the case of a GF(2^{191}) binary field.

The hardware implementation of the MULGF2 operation is possible on a polynomial multiplier or even on a unified dual-field multiplier [48], able to manage integers and polynomials at the cost of a small increase in area and delay. The performance benefits are impressive because the hardware MULGF2 can be implemented to run in only one or two cycles, against 190 of the software emulation.

Table 9.3 shows the experimental results of the performance of the multiplication algorithm, as well as of the EC scalar multiplication operation, for both standard ISA and algorithm-specific ISE.

Table 9.3 Execution time (cycles) of the various multiplication algorithms on GF(p) and GF(2^m) for the standard ISA and with algorithm-specific ISE.

Operation	GF(p) Schoolbook	GF(p) Comba	GF(2^m) Shift-XOR	GF(2^m) Word-level
Field mul. (std ISA)	629	827	2758	7848
Field mul. (with ISE)	485	**441**	2151	456
Point mul. (std ISA)	2160k	2840k	4050k	10420k
Point mul. (with ISE)	1670k	1470k	3280k	**870k**

The table shows very interesting results for ECC on both prime and binary fields at the same security level: GF(p), with $p = 2^{192} - 2^{64} - 1$, and GF(2^{191}). The fastest multiplication operation on the field is achieved for GF(p) with the described ISE (441 cycles), even if the result for GF(2^m) and custom ISE is very close to it (456 cycles).

However, the results for the EC scalar multiplication k[P], which is significant at application level, show a different trade-off. GF(2^m) results significantly outperform GF(p) ones (870 vs. 1470 kilo cycles), indicating the complex interaction between architectural features of the target system, the instruction set architecture and its extensions, the field-level, and the EC-level algorithms.

In a previous paper [23], Großshädl et al. highlight the useful ISEs for prime (GF(p)) and binary (GF(2^m)) finite fields and present some additional instructions to the ones that we have already described. For instance, on GF(p), they propose *M2ADDU Rs, Rt* which is a slightly modified multiply-and-accumulate (*MADDU*) which doubles the partial product before accumulation. The additional complexity is negligible because doubling on integers requires only a left shift. This instruction is very useful in multiple-precision squaring of integers.

The *ADDAU Rs, Rt* adds the two input registers and accumulates the result on the *HI*/*LO* register pair. This is useful to support multiple-precision addition and reduction modulo a generalized Mersenne prime.

The *SHA* performs a 32-bit right shift of the *HI*/*LO* accumulator, in order to move the value in *HI* into the *LO* register.

For GF(2^m), they propose *MULGF2 Rs, Rt* and *MADDGF2 Rs, Rt*, which perform the word-level polynomial multiplication and multiply-and-accumulate, respectively.

As a training exercise, the reader is encouraged to study the paper [21], which presents a study on possible ISEs in the case of optimal extension fields (OEF). The approach has similarities with the other ones already discussed but has also some differences originating from the specific features of OEFs.

A similar study has been conducted by Eberle et al. in [10] on an 8-bit ATmega128 8 MHz processor. The authors show that simple extensions of the available datapath suffice to efficiently support ECC over GF(2^m) and, in addition, to outperform GF(p). The extensions include a dual-field multiplier for both integers and

polynomials using the carry-save adder (CSA) tree structure. The proposed ISEs comprise *MULX* and *MULACCX* instructions, which are designed so as to maintain compatibility with existing instructions (*i.e.*, two operands only), and the existing datapath (*i.e.*, up to two source operands can be read and two destination operands can be written by a functional unit).

- *MULX* R_d, R_r instruction performs the polynomial multiplications between R_d and R_r and puts the doubleword result into the R1:R0 register pair.
- *MULACCX* R_d, R_r instruction performs a multiply-accumulate operation with extended carry. In particular, $R_d \leftarrow bits[7:0]((R_r \otimes R_c) \oplus XC \oplus R_d)$, and $XC \leftarrow bits[15:8]((R_r \otimes R_c) \oplus XC \oplus R_d)$, where R_c is an implicit architectural register and XC is a non-architectural register. Note that R_c must be loaded prior to executing *MULACCX*.

Figure 9.16 shows in grey the words that can be fruitfully managed by a *MULACCX* instruction. In fact, the basic idea is to reuse in the following *MULACCX* the XC part of the result obtained by the previous one. In particular, the instruction would be used in this way: *MULACCX* R_d, R_r, where $R_d = B_n$, $R_r = c_{n+p}$, would calculate $c_{n+p} = (a_p \otimes b_n) \oplus (bits[15:8](a_p \otimes b_{n-1}) \oplus c_{n+p})$, where R_c holds a_p and XC holds $bits[15:8](a_p \otimes b_{n-1})$. The example in the figure highlights when $p = 1$ and $n = 3$.

Note that the XC register is not explicitly accessible to the programmer. When XC must be saved and restored, for instance because of a system call, its value can be retrieved using a dummy *MULACCX* $R_d = 0$, $R_r = 0$ instruction to move XC value to R_d. The dual operation can be performed by a similar technique (the reader should look for it as a useful training exercise).

The performance originating from such ISEs are significant: 0.40 and 0.29 seconds for EC scalar multiplication on GF(2^{163}) using *MULX* and *MULACCX*,[1]

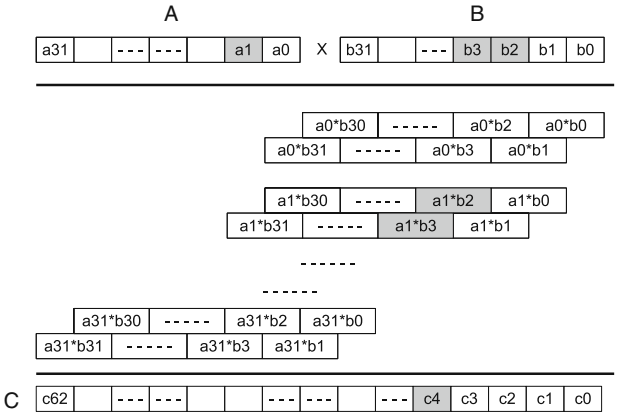

Fig. 9.16 Multiple-precision multiplication in GF(2^m). *MULACCX* instruction can speed up the management of the data highlighted in grey.

[1] executing in one cycle.

respectively, compared to 4.14 seconds of the unmodified ISA. Moreover, the performance of k[P] for GF(p) on a 160-bit prime is 0.81 seconds.

As a comparison, RSA performance for the same security level (1024-bit), are 0.43 and 10.99 seconds for public-key and private-key operations, respectively. Moreover, switching to higher security level, 2048-bit RSA and 233-bit ECC, the speedup given by the ISE over the standard ISA is almost the same (about 14x) but the performance gap between ECC and RSA widens. In fact, in this case, ECC with *MULX* and *MULACCX* runs in 1.12 and 0.81 seconds, respectively, against 10.98 seconds of the unmodified ISA, while RSA public-key and private-key timings are 1.94 and 83.26 seconds, respectively.

Bartolini et al. in [1] analyze the performance of some ECC applications (Diffie-Hellman key-exchange, digital-signature algorithm, El-Gamal encryption/decryption) on a 32-bit ARM-based Intel XScale processor. The performances are investigated through SimpleScalar, [5] a cycle-accurate simulator of the target processor which allows one to analyze the behavior of all internal modules of the processor, as well as of the memory hierarchy (*i.e.*, caches, memory bus, RAM). A specifically modified version of the simulator allowed to highlight the time spent in each of the finite-field, elliptic-curve and other operations.

Figure 9.17 shows that 32-bit word-level polynomial multiplication (*mr_mul*2) takes up a significant fraction of the time on the considered benchmarks. In particular, from 18% in the digital signature benchmarks (*ecDSsign* and *ecDSver*), which, however, spend more than 64% of the time in hash generation and file reading, up to 54% in Diffie-Hellman key-exchange protocol (*ecDH*).

Operation *mr_mul*2 is the software procedure that performs a 32-bit word-level polynomial multiplication, similar to the MULGF2 operation cited by Koç and Acar in [34]. The procedure is optimized through loop unrolling and by the use of a small look-up table that speeds up the shift-and-XOR approach. However, on the given architecture and with the particular library (MIRACL [53]), it takes about 400 dynamic instructions (roughly 500 cycles), which correspond to about 12 instructions per bit. The library is able to support different operand sizes and, therefore, is not fully optimized for a particular key size.

Apart from *mr_mul*2, the other operations highlighted in Figure 9.17 are:

- reduce2: GF(2^m) modulo reduction
- mr_sqr2 and square2: word-level and GF(2^m) squaring, respectively
- karmul2: manages Karatsuba algorithm for GF(2^m) polynomial multiplication
- mr_bottom4: a base case of Karatsuba algorithm
- add2: GF(2^m) addition, *i.e.* XOR operation over m-bit polynomials
- numbits: bit count
- hash and file read: hash algorithm and file-reading activities
- other: other functions for the management of finite-field values

The figure highlights that, among the cryptographic operations, the most time-consuming one is *mr_mul*2, and then, with a notably smaller weight, the modular reduction, word-level squaring and the multiprecision management of Karatsuba multiplication.

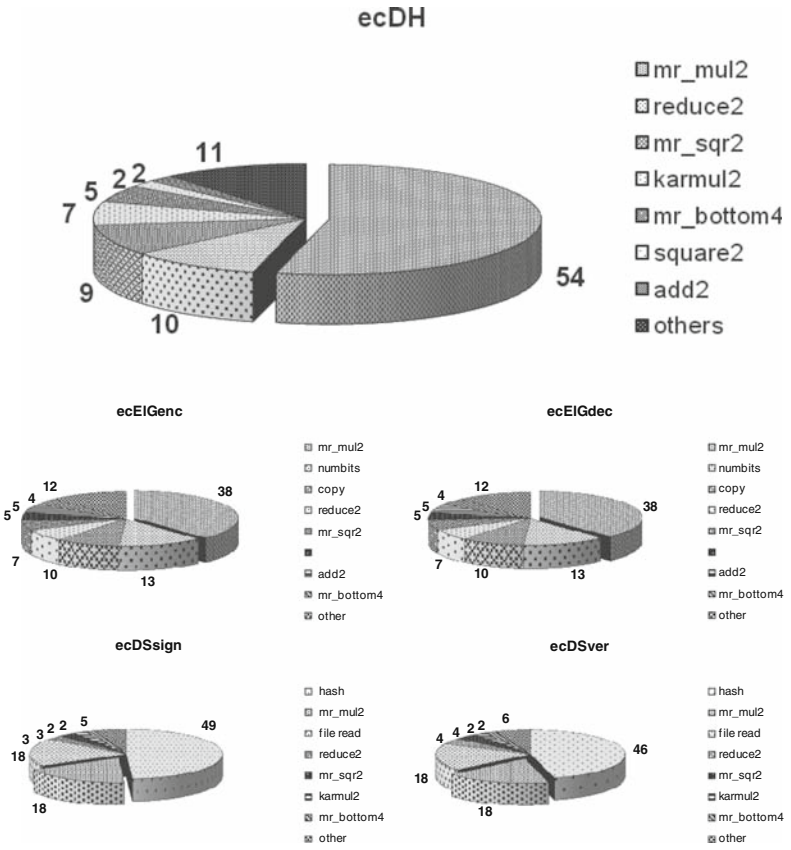

Fig. 9.17 Execution time breakdown for various EC benchmarks. Legend items are positioned clockwise on the pies and the word-level 32-bit polynomial multiplication (*mr_mul*2) is highlighted.

The authors evaluate the effects of including a 32-bit word-level polynomial multiplication instruction, named *MULGF*, to support *mr_mul*2 in hardware through a specific multiplier unit. A specific multiplier could be used, simpler than the integer one, or a dual-field multiplier [48] for both polynomials and integers. The latency of the multiplier is modeled conservatively, equal to the integer multiplier: three cycles, which is reasonable because the polynomial multiplication circuit can be quicker than the integer one because of the lack of carry propagation. The software procedure for word-level polynomial multiplication was substituted by the *MULGF* instruction throughout the ECC library using assembler inlining, and the GCC cross-compiler was modified for managing the additional *MULGF* instruction. In this way, the compiler is able to apply a number of optimizations to the extended instruction flow (*e.g.*, instruction scheduling, register allocation, generation of machine code).

Table 9.4 Effects of the adoption of *MULGF* instruction on both execution time and on the number of executed instructions.

Benchmark	Execution time %	Dyn. instruction %
ecDH	-54	-55
ecElGenc	-39	-48
ecElGdec	-35	-37
ecDSsign	-17	-19
ecDSver	-17	-19

In this way, the 400 dynamic instructions for executing *mr_mul*2 are collapsed into only one *MULGF* instruction and, correspondingly, the 500 cycles of the software *mr_mul*2 are distilled into the *MULGF* latency of three cycles.

For GF(2^{233}), Table 9.4 shows that the resulting improvement in execution time is more significant for Diffie-Hellman (54% in number of instructions and 55% in execution time) and El-Gamal algorithms (48% and 37% less instructions for encryption and decryption, respectively, corresponding to 39% and 35% less execution time), where 32-bit polynomial multiplication is used more. The improvement for digital signature algorithm is less evident (19% in instruction number and 17% in execution time for the same key length) because of the included hash time.

Apart from the plain performance speedup, the authors investigate on the origins of the performance and highlight the effects of the ISE on the instruction-level parallelism that the processor is able to exploit. Figure 9.18 shows the cycles per instruction (CPI) performance metric in the case of a 1Byte I-Cache and D-Cache organization. The CPI is split into the cycles spent for actual execution of the instructions (CPI-processor) and the cycles spent in waiting for memory (CPI-memory): operands (load/stores) or instructions (fetch). In addition to reducing the number of dynamically executed instructions, the ISE allows one to reduce the CPI, which means that the processor executes the remaining instructions faster. In particular, the CPI-memory quote is significantly reduced due to a more efficient behavior of the caches. This is due to the smaller footprint of the algorithm in terms of executed instructions (*i.e.*, less instruction fetches) and to the reduced number of spills from registers to memory in the computation of the *mr_mul*2 operation (*i.e.*, less load/stores).

Kumar and Paar in [35] evaluate the usage of a 163-bit *full-width* coprocessor for performing modular multiplication on a simple 8-bit processor. They propose that the additional hardware is closely coupled with the ALU of the processor, reducing the interface circuitry and aiming for efficient implementation. The circuit is designed to deliver high performance on GF(2^{163}) multiplications, but it cannot be as flexible as word-level ISE approaches, which can easily accommodate to different-sized fields.

The proposed extension is able to directly access the data-RAM so that the main processor can avoid transfer of data to/from memory for using the unit. The proposed 163-bit multiplier unit takes 42 cycles to execute, which become 193 cycles including all the set up and control overhead.

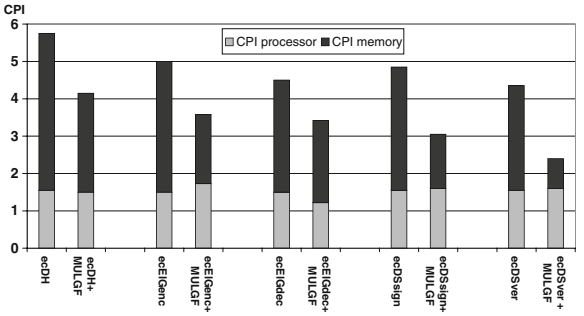

Fig. 9.18 Cycles Per Instructions (CPI) for the considered EC benchmarks for original and MULGF extended instruction set.

The unit is managed through an extended instruction which triggers the functional unit to load the operands (21 8-bit words) directly from memory, to execute the multiplication and to store back the result to memory. The address of the operands and of the result should be prepared into three fixed processor registers. During the multiplication, the processor must poll the unit for understanding when the operation has finished. An alternative is to implement and program the unit to raise an interrupt upon completion.

The proposal delivers a speedup of more than 30x over the original software implementation and allows the EC scalar multiplication (k[P]) to execute the operation in 169ms on a 4 MHz 8-bit microcontroller. In addition, the proposed ISE shrinks down the code and data size of the k[P] to 2048 and 273 bytes, respectively, whereas the original application required 8208 and 358 bytes, respectively.

Batina et al. in [29] propose to extend an 8-bit 8051 processor with a full-width multiply-and-accumulate unit for GF(2^{83}) which is interfaced toward the processor through an 8-bit bus. The unit width is suitable for hyper-elliptic curve cryptography (HECC) on GF(2^{83}) binary field.

The unit presented here is equipped with three full-width registers for holding modular multiplication operands (A, B) and result (C). Such registers are write (A,B) and read (C) in 8-bit chunks.

The unit is able to perform full-width multiplication and addition operations, as well as move operations between C and B register. In this way, multiply and accumulate can be performed in two steps (multiplication and accumulation) reusing part of the values already present in the unit.

The performance time for executing a modular multiplication with the new unit is 28.2 K-cycles. Modular addition is again 28.2 K-cycles, even if the plain circuit takes 83 cycles for a multiplication and 1 cycle for the addition. This is because much of the time is spent in I/O operations to transfer the operands to/from the unit. In software, the corresponding results are 650 K-cycles and 38 K-cycles, respectively. The multiply-and-accumulate operation using the new unit takes advantage of the already available multiplication result and runs in only 30.5 K-cycles. Also, in HECC, modular multiplication is the most time-consuming operation and

therefore, ISEs for modular multiplication can be very useful, especially in constrained devices.

It is interesting to highlight the proposal from Grabbe et al. [19], which is not exactly an ISE approach but a processor proposal for finite-field arithmetic and ECC. In fact the authors propose a very long instruction word (VLIW) processor with a number of full-width units able to work on 233-bit field values: adder, multiplier and squarer.

The adder takes 1 cycle to process two or three operands. The multiplier latency is 9 cycles but it is pipelined so that every two cycles a new operand pair can be fed to the unit. The latency of the squarer is one cycle.

Two independent register files hold four 233-bit registers each. We will not go into the details of the architecture of the processor here, because it goes beyond the scope of this chapter, but we want to highlight a design choice that allowed one to introduce some high-level instructions for ECC.

In fact, the proposed processor is able to manage the high-level EC operations through specific instructions which are executed via a microcoded approach. EC point addition, doubling, inversion, and the scalar multiplication are supported natively.

Obviously, there is no hardware circuit that executes such instructions as a whole. Each of these instructions is executed through a sequence of finite-field operations, supported by the processor functional units, which are orchestrated according to a microprogram stored in a control unit.

In other words, the processor supports two kinds of instructions, the ones directly executed by a specific unit (*e.g.*, field squaring, multiplication and addition) and others, more complex, which are executed through a *program* that uses the available processor units.

This can be an interesting approach to raise the level of abstraction of the operations managed by the processor autonomously. Certainly, this also leads to a less flexible architecture because the microcode that implements the high-level algorithms is fixed.

However, the availability of the lower-level instructions, allows one to program explicitly alternative algorithms also for EC operations, even if in a less efficient way than with microcode.

The performance results of this full-width microcoded approach are impressive: 31 and 42 clock cycles to execute 233-bit EC-doubling and EC-addition, respectively, which translate into about 12 K-cycles for an average EC scalar multiplication. The microcode implements the EC scalar multiplication using the add-and-double technique.

9.4 Exercises

1. Discuss the benefits of elliptic curves for public-key cryptography and show an instruction-set extension suited for a cryptographic system based on this approach.

2. Analyze the performances of the pencil-and-paper and Comba's methods for polynomial multiplication discussed in Section 9.3.3. The idea is to implement the algorithms in a high-level language and then inspect the code. Assume that assignments and additions/subtractions on integers and logic operations (*e.g.*, XOR) take 1 cycle, memory accesses (read/writes) take 3 cycles. Then, consider the following variant: up to 8 integer values (variables) can be maintained in the processor registers without the need of update/reread the memory location of the corresponding variable up to when the register has to be reassigned a new value. This should refine the estimation of the required memory access time.

9.5 Projects

1. Implement a multiprecision addition and multiplication procedure for 1024-bit integers and measure the performance through repeated random testing: generate random numbers for the operands, *e.g.* using the random number generator of the adopted high-level language (*e.g.*, outputting 32-bits at a time) to fill the 1024-bit operands.
2. Implement a simple symmetric block-cipher that uses a 128-bit symmetric key and works as follows. The plain text is encrypted 128-bit (block) at a time doing an XOR operation with the key. The decryption process is done exactly in the same way, using the same key. The key for encrypting a block is obtained by the key of the previous block multiplying it by 3 and getting the result mod 2^{128}. The first key is the initial key. Small plaintexts and the last block of bigger ones should be managed properly. Implement the solution in a high-level language and then try to apply some optimizations using the assembler on the target machine. For instance, exploit the carry flag and add with carry instructions for multiprecision additions.
3. Implement the AES block-cipher from the documentation[2] using a high-level language and analyze the performance of the main high-level operations. A possible way is to evaluate the performance of each of them separately and then count the number of times they are called in an AES encryption/decryption. In this way, an estimate can be drawn for the time spent in each operation by AES. Then, discuss the performance benefits from possible ISEs that accelerate specific operations.
4. Implement a multiprecision polynomial library GF(2^{163}), adopting the NIST irreducible polynomial ($p(x) = x^{163} + x^7 + x^6 + x^3 + 1$) and implementing addition (XOR), reduction, squaring and multiplication. Try to implement the field operations relying on word-level operations (*e.g.*, 163-bit multiplication, *fieldMult*, done in 32-bit chunks, *wordMult*.) which are implemented in specific functions. Measure the time spent in the various word-level functions when executing the high-level field operations and analyze the word-level function that take up most

[2] see, e.g., http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

of the time and thus would deserve a specific ISE. Consider the following relative number of invocations for the high-level library functions, normalized to 100 overall invocations: modular multiplication 20, modular squaring 60 and addition 20 of the time.

References

1. S. Bartolini, I. Branovic, R. Giorgi, and E. Martinelli. A performance evaluation of arm isa extension for elliptic curve cryptography over binary finite fields. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pp. 238–245, 27-29 Oct. 2004. 10.1109/SBAC-PAD.2004.5.
2. G. M. Bertoni, L. Breveglieri, F. Roberto, and F. Regazzoni. Speeding up AES by extending a 32-bit processor instruction set. In *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, pp. 275–282, Sept. 2006. 10.1109/ASAP.2006.62.
3. I. Branovic, R. Giorgi, and E. Martinelli. A workload characterization of elliptic curve cryptography methods in embedded environments. *ACM SIGARCH Computer Architecture News*, 32(3):27–34, June 2004. ISSN 0163-5964. http://doi.acm.org/10.1145/1024295.1024299.
4. M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the nist elliptic curves over prime fields. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pp. 250–265, London, UK, 2001. Springer-Verlag. ISBN 3-540-41898-9.
5. D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997. ISSN 0163-5964.
6. J. Burke, J. McDonald, and T. Austin. Architectural support for fast symmetric-key cryptography. *SIGPLAN Not.*, 35(11):178–189, 2000. ISSN 0362-1340. http://doi.acm.org/10.1145/356989.357006.
7. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.
8. Counterpane Internet Security Inc. *The blowfish encryption algorithm*, 1993. http://www.counterpane.com/blowfish.html.
9. J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2.
10. H. Eberle, A. Wander, N. Gura, Sheueling Chang-Shantz, and V. Gupta. Architectural extensions for elliptic curve cryptography over gf(2/sup m/) on 8-bit microprocessors. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pp. 343–349, 23-25 July 2005. 10.1109/ASAP.2005.15.
11. H. Eberle, N. Gura, S. C. Shantz, V. Gupta, L. Rarick, and S. Sundaram. A public-key cryptographic processor for rsa and ecc. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, pp. 98–110,

Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2226-2. http://dx.doi.org/10.1109/ASAP.2004.6.

12. A. J. Elbirt. Fast and efficient implementation of AES via instruction set extensions. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pp. 396–403, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2847-3. http://dx.doi.org/10.1109/AINAW.2007.182.
13. Federal Information Processing Standards Publication 46-1. *Data encryption standard (DES)*, 1988.
14. Federal Information Processing Standards Publication 46-3. *Data encryption standard (DES) - tdea*, 1999.
15. Federal Information Processing Standards Publication 197. *Specification for the advanced encryption standard (AES)*, 2001.
16. A. M. Fiskiran and R. B. Lee. Evaluating instruction set extensions for fast arithmetic on binary finite fields. In *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004*, pp. 125–136. IEEE Computer Society, 2004. ISBN 0-7695-2226-2.
17. A. M. Fiskiran and R. B. Lee. Performance scaling of cryptography operations in servers and mobile clients. In *Proceedings of the Workshop on Building Block Engine Architectures for Computer Networks (BEACON)*, 2004.
18. J. R. Goodman. *Energy scalable reconfigurable cryptographic hardware for portable applications*. PhD thesis, Massachusetts Institute of Technology, Deptartment of Electrical Engineering and Computer Science, 2000.
19. C. Grabbe, M. Bednara, von zur Gathen, J. Shokrollahi, and J. Teich. A high performance vliw processor for finite field arithmetic. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 6pp., 22–26 April 2003. 10.1109/IPDPS.2003.1213351.
20. J. Großschädl and G.-A. Kamendje. Optimized RISC architecture for multiple-precision modular arithmetic. In *International Conference on Security in Pervasive Computing, LNCS*, 2003.
21. J. Großschädl, S. S. Kumar, and C. Paar. Architectural support for arithmetic in optimal extension fields. In *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 111–124, 2004. 10.1109/ASAP.2004.1342463.
22. J. Großschädl and G.-A. Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2^m). In E. Deprettere, S. Bhattacharyya, J. Cavallaro, A. Darte, and L. Thiele, editors, *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 455–468. IEEE Computer Society Press, 2003. ISBN 0-7695-1992-X.
23. J. Großschädl and E. Savaş. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pp. 133–147. Springer Verlag, 2004. ISBN 3-540-22666-4.

24. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. http://dx.doi.org/10.1109/WWC.2001.15.
25. D. Hankerson, J. López, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *International Workshop on Cryptographic Hardware and Embedded Systems - CHES*, pp. 1–24, 2000.
26. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN 038795273X.
27. Y. Hilewitz and R. B. Lee. Performing advanced bit manipulations efficiently in general-purpose processors. In *IEEE Symposium on Computer Arithmetic*, pp. 251–260, 2007.
28. Y. Hilewitz, Z. Jerry Shi, and R. B. Lee. Comparing fast implementations of bit permutation instructions. In *Proceedings of the 38th Annual Asilomar Conference on Signals, Systems, and Computers*, pp. 1856–1863, "November" 2004.
29. A. Hodjat, L. Batina, D. Hwang, and I. Verbauwhede. Hw/sw co-design of a hyperelliptic curve cryptosystem using a microcode instruction set coprocessor. *Integr. VLSI J.*, 40(1):45–51, 2007. ISSN 0167-9260. http://dx.doi.org/10.1016/j.vlsi.2005.12.011.
30. Intel. *IA-64 Architecture Software Developer's Manual*, May 1999.
31. Intel. *Ia-32 intel architecture software developers manual volume 1: Basic architecture*, 2004.
32. Intel. *Intel SSE4 programming reference*, July 2007.
33. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48: 203–209, ISSN 0025–5718 1987.
34. Ç. K. Koç and T. Acar. Montgomery Multiplication in GF(2^k). *Des. Codes Cryptography*, 14(1):57–69, 1998. ISSN 0925-1022. http://dx.doi.org/10.1023/A:1008208521515.
35. S. S. Kumar and C. Paar. Reconfigurable instruction set extension for enabling ecc on an 8-bit processor. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *FPL*, volume 3203 of *Lecture Notes in Computer Science*, pp. 586–595. Springer, 2004. ISBN 3-540-22989-2.
36. X. Lai. *On the Design and Security of Block Ciphers*. Hartung-Gorre Verlag, 1992.
37. R. B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, January 1989.
38. R. B. Lee. Subword parallelism with MAX-2: Accelerating media processing with a minimal set of instruction extensions supporting efficient subword parallelism. *IEEE Micro*, 16(4):51–59, August 1996. ISSN 0272-1732.
39. R. B. Lee, Z. Shi, and X. Yang. Cryptography efficient permutation instructions for fast software. *IEEE Micro*, 21(6):56–69, 2001.
40. J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2^m) without precomputation. In *CHES: International Workshop on Cryptographic Hardware and Embedded Systems, CHES, LNCS*, 1999.

41. J. López and R. Dahab. High-speed software multiplication in f2m. In *INDOCRYPT '00: Proceedings of the First International Conference on Progress in Cryptology*, pp. 203–212, London, UK, 2000. Springer-Verlag. ISBN 3-540-41452-5.
42. J. P. McGregor and R. B. Lee. Architectural enhancements for fast subword permutations with repetitions in cryptographic applications. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD '01)*, pp. 453–461, Washington - Brussels - Tokyo, September 2001. IEEE. ISBN 0-7695-1200-3.
43. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Norwell, MA, USA, 1994. ISBN 0792393686. Foreword By-Neal Koblitz.
44. V. S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, pp. 417–426, Santa Barbara, California, USA, August 1985.
45. National Institute of Standrds and Technology. *Fips-197: Advanced encryption standard*, November 2001. http://csrc.nist.gov/publications/fips/.
46. National Institute of Standrds and Technology. *Fips-180-2: Secure hash standard*, August 2002. http://csrc.nist.gov/publications/fips/.
47. C. Paar. The future of the art of cryptographic implementations. In *Position Statement for the STORK Workshop*, Brussels, Nov. 2002.
48. E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields gf(p) and gf(2^m). In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 277–292, London, UK, 2000. Springer-Verlag. ISBN 3-540-41455-X.
49. Princeton Architecture Laboratory for Multimedia and Security (PALMS). *Pax project*, 2003. http://palms.ee.princeton.edu/PAX.
50. C. E. Shannon. Communication theory of secrecy systems. *Bell Systen Technicl Journal*, 28:656–715, October 1949.
51. Z. Shi and R. B. Lee. Bit permutation instructions for accelerating software cryptography. In *ASAP '00: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 138, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0716-6.
52. Z. Shi, X. Yang, and R. B. Lee. Arbitrary bit permutations in one or two cycles. In *ASAP '03: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 237. IEEE Computer Society, 2003. ISBN 0-7695-1992-X.
53. S. Software. *MIRACL: Multiprecision Integer and Rational Arithmetic C/C++ Library*, 1988. http://www.shamus.ie/.
54. S. Tillich and J. Großschädl. Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography. In Marina Gavrilova, Youngsong Mun, David Taniar, Osvaldo Gervasi, Kenneth Tan, and Vipin Kumar, editors, *Computational Science and Its Applications - ICCSA 2005*, volume 3481 of *Lecture Notes in Computer Science*, pp. 665–675. Springer, 2005.
55. S. Tillich and J. Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In Louis Goubin and Mitsuru Matsui,

editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10–13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pp. 270–284. Springer, 2006.

56. A. K. Verma, L. Pozzi, P. Ienne, S. Tillich, and J. Großschädl. When instruction set extensions change algorithm design: A study in elliptic curve cryptography. In *4th Workshop on Application-Specific Processors (WASP 2005)*, p. 29, Jersey City, NJ, USA, September 2005.
57. L. Wu, C. Weaver, and T. Austin. Cryptomaniac: a fast flexible architecture for secure communication. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 110–119, New York, NY, USA, 2001. ACM Press. ISBN 0-7695-1162-7. http://doi.acm.org/10.1145/379240.379256.
58. X. Yang and R. Lee. Fast subword permutation instructions using omega and flip network stages. In *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, pp. 15–22, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0801-4.
59. X. Yang, M. Vachharajani, and R. Lee. Fast subword permutation instructions based on butterfly networks. In *Proceedings of SPIE, Media Processor*, pp. 80–86, January 2000.
60. P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, 1995.

Chapter 10
FPGA and ASIC Implementations of AES

Kris Gaj and Pawel Chodowiec

10.1 Introduction

In 1997, an effort was initiated to develop a new American encryption standard to be commonly used well into the next century. This new standard was given a name AES, *Advanced Encryption Standard*.

A new algorithm was selected through a contest organized by the National Institute of Standards and Technology (NIST). By June 1998, 15 candidate algorithms had been submitted to NIST by research groups from all over the world. After the first round of analysis was concluded in August 1999, the number of candidates was reduced to final five. In October 2000, NIST announced its selection of *Rijndael* [7] as a winner of the AES contest. The official standard was published in November 2001 as FIPS (Federal Information Processing Standard) number 197 [1].

The primary criteria used by NIST to evaluate AES candidates included security, efficiency in software and hardware, and flexibility. In the absence of any major breakthroughs in the cryptanalysis of the final five candidates, and because of the relatively inconclusive results of their software performance evaluations, hardware efficiency evaluations presented during the third AES conference provided a very substantial quantitative measure that clearly differentiated AES candidates among each other [9, 10, 12, 17, 21, 42]. The importance of this measure was reflected by a survey performed among the participants of the AES conference, in which the ranking of the candidate algorithms coincided very well with their relative speed in hardware [16, 18].

The AES evaluation process resulted in the first efficient hardware architectures for AES. The university groups contributed first implementations of AES based on FPGAs (field programmable gate arrays) [5, 9, 11, 18]. The National Security Agency group and industry groups provided the first implementations targeting ASICs (application-specific integrated circuits) [21, 42].

George Mason University
e-mail: {kgaj,pchodow1}@gmu.edu

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_10,

© Springer Science+Business Media, LLC 2009

A substantial progress in the development of the new architectures for AES has been made after the conclusion of the contest, as a result of focusing research efforts on a single secret-key encryption standard. This progress proceeded in several major directions.

One direction was the development of high-speed, highly pipelined architectures for non-feedback cipher modes. This direction led to the development of AES implementations operating with the speeds of tens of Gigabits per second [22, 23, 25, 26, 29, 32, 34–36, 41]. The second direction was the development of compact architectures for AES, optimized for the minimum area. This effort led to the emergence of architectures with 64-, 32-, and even 8-bit data paths [2, 6, 19, 20, 27, 33, 45].

The third direction was the optimization of basic operations of AES, including logic-only implementation of *SubBytes* [3, 4, 28–31, 33, 44] and optimizations and decompositions of the *MixColumns* and *InvMixColumns* transformations [6, 14, 15, 43]. Still, a different direction was the development of new architectures for the entire encryption/decryption unit [13].

In this chapter, we will review the AES algorithm from the point of view of knowledge required for efficient hardware implementations. We will then describe several alternative ways of implementing all basic operations and the entire cipher. We will conclude with our recommendations regarding the optimum choice of particular design options and the entire hardware architecture for AES depending on requirements of a particular application.

10.2 AES Cipher Description

10.2.1 Basic Features

AES is a symmetric-key block cipher. AES operates on 128-bit data blocks and accepts 128-, 192-, and 256-bit keys. It is an iterative cipher, which means that both encryption and decryption consist of multiple iterations of the same basic round function, as shown in Figure 10.1.

In each round, a different *round* (or *internal*) key is being used. In AES, the number of cipher rounds depends on the size of the key. It is equal to 10, 12, or 14 for 128-, 192-, or 256-bit keys, respectively.

Based on the internal structure of a round function, AES belongs to the group of SP-network block ciphers. This means that the main transformations employed in this cipher are substitutions and permutations applied to all bits of data block in every round. Data blocks are internally represented in a square form, called State, which is shown in Figure 10.2. In this diagram, each field represents one byte of data.

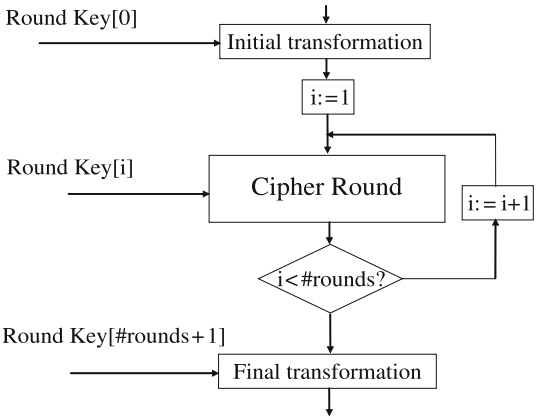

Fig. 10.1 Flowchart of a generic iterative cipher.

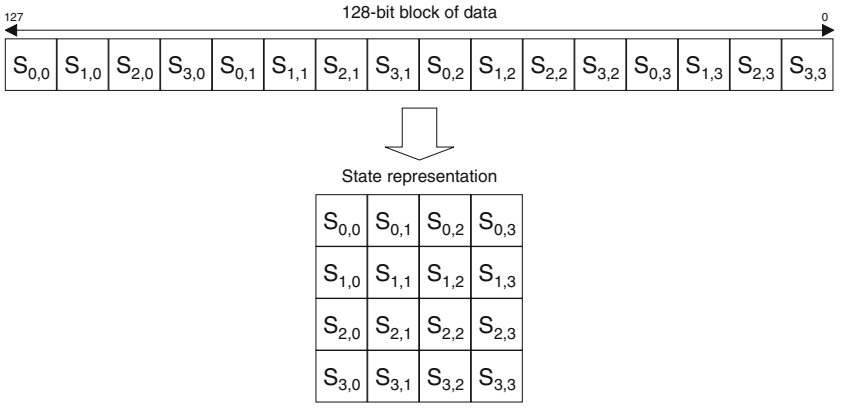

Fig. 10.2 State representation of 128-bit data blocks.

10.2.2 Round Operations

AES encryption round employs consecutively four main operations: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. Since Rijndael is an SP-network cipher, it requires an inversed version of all transformations for decryption. These inverse transformations are called *InvSubBytes*, *InvShiftRows*, *InvMixColumns*, and *InvAddRoundKey*. Please note that the last transformation of an encryption round, *AddRoundKey*, is equivalent to a bitwise XOR and therefore is an inverse of itself. The structure of encryption and decryption rounds is shown in Figure 10.3.

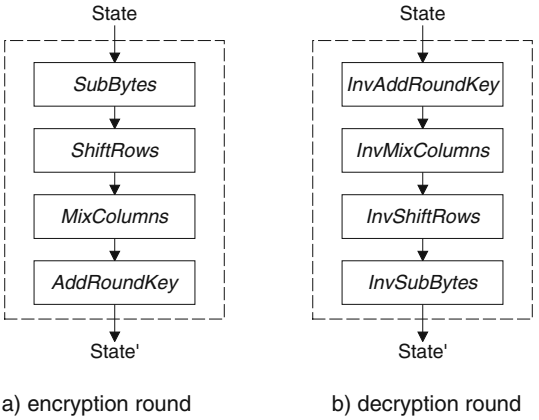

Fig. 10.3 Structure of AES encryption and decryption round.

10.2.2.1 Operations in the Galois Field $GF(2^8)$

Two of the AES round operations, *SubBytes* and *MixColumns*, rely on operations in the Galois field $GF(2^8)$. Each element of this field can be treated as either an 8-bit string (in the binary or hexadecimal representation) or as a polynomial of degree seven or less, with coefficients in $\{0,1\}$ (polynomial basis representation). The coefficients of a polynomial are equal to the respective bits of the binary representation. For example, $\{03\}$ in hexadecimal is equivalent to $\{0000\ 0011\}$ in binary, and to

$$c(x) = 0 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x + 1 \cdot 1 = x + 1 \qquad (10.1)$$

in the polynomial basis representation. The multiplication of elements of $GF(2^8)$ in AES is accomplished by multiplying the corresponding polynomials modulo a fixed irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

For example, multiplying a variable element $a = a_7a_6a_5a_4a_3a_2a_1a_0$ by a constant element $\{03\}$ is equivalent to computing

$$\begin{aligned} b(x) &= b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \\ &= (a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot (x+1) \\ &\quad \bmod (x^8 + x^4 + x^3 + x + 1) \end{aligned} \qquad (10.2)$$

After several simple transformations

$$\begin{aligned} b(x) \quad &= (a_7 + a_6) \cdot x^7 + (a_6 + a_5) \cdot x^6 + (a_5 + a_4) \cdot x^5 + (a_7 + a_4 + a_3) \cdot x^4 \\ &\quad + (a_7 + a_3 + a_2) \cdot x^3 + (a_2 + a_1) \cdot x^2 + (a_7 + a_1 + a_0) \cdot x + (a_7 + a_0) \end{aligned}$$

where "+" represents an addition modulo 2, i.e., an XOR operation. Each bit of a product b can be represented as an XOR function of at most three variable input bits, e.g., $b_7 = (a_7 + a_6)$, $b_4 = (a_4 + a_3 + a_7)$, etc.

10.2.2.2 *SubBytes* and *InvSubBytes*

The *SubBytes* operation transforms individual bytes of the internal state as shown in Figure 10.4. Internally, it is composed of two basic operations:

1. Multiplicative inversion in the Galois field $GF(2^8)$ with the reduction polynomial $m(x)$ specified by Equation (10.3). Element $\{00\}$ is mapped onto itself.

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{10.3}$$

2. Affine transformation over $GF(2)$:

$$b'_i = b_i + b_{(i+4)\ mod\ 8} + b_{(i+5)\ mod\ 8} + b_{(i+6)\ mod\ 8} + b_{(i+7)\ mod\ 8} + c_i \tag{10.4}$$

where byte c has value $\{63\}$ or $\{01100011\}$.
Equation (10.5) shows the affine transformations in the matrix form.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \\ 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{10.5}$$

The inversed version of *SubBytes*, called *InvSubBytes*, employs identical multiplicative inversion in $GF(2^8)$ and an inversed affine transformation. Equation (10.6) shows the inverse affine transformations in the matrix form:

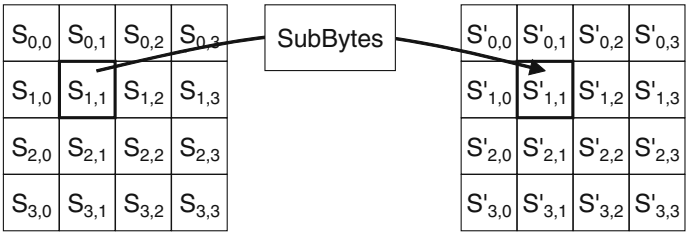

Fig. 10.4 Application of *SubBytes* to State.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0\,0\,1\,0\,0\,1\,0\,1 \\ 1\,0\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,0\,1\,0\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,1\,0\,0\,1 \\ 1\,0\,0\,1\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{10.6}$$

The corresponding equation in $GF(2)$ is given in (10.7) as

$$b'_i = b_{(i+2)\ mod\ 8} + b_{(i+5)\ mod\ 8} + b_{(i+7)\ mod\ 8} + d_i \tag{10.7}$$

The internal structure of *SubBytes* and *InvSubBytes* is shown in Figure 10.5.

10.2.2.3 *ShiftRows* and *InvShiftRows*

The *ShiftRows* and *InvShiftRows* cyclically shift three bottom rows of the State by a different number of positions, one, two, and three, respectively, as shown in Figure 10.6. Without those operations all-round transformations would be limited only to the columns of the State.

10.2.2.4 *MixColumns* and *InvMixColumns*

MixColumns and *InvMixColumns* operations are defined over 4-byte words that represent a column of the State as shown in Figure 10.7. These 4-byte words are considered as polynomials (of degree of at most 3) with coefficients in $K = GF(2^8)$, defined in the ring of polynomials $K[X]$ modulo $M(X) = X^4 + 1$ and denoted as

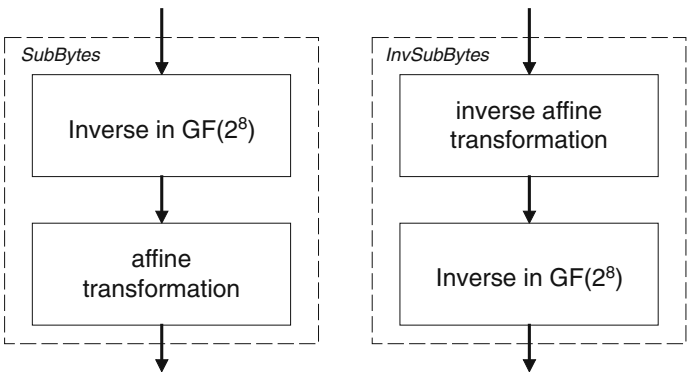

Fig. 10.5 Composition of *SubBytes* and *InvSubBytes*.

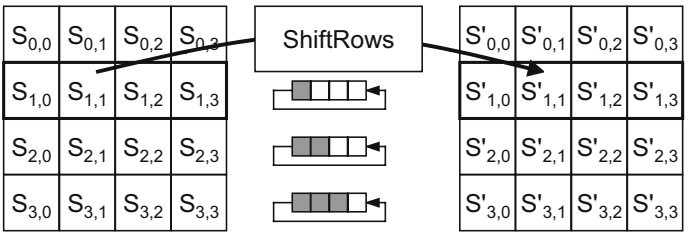

Fig. 10.6 *ShiftRows* transforming rows of a State.

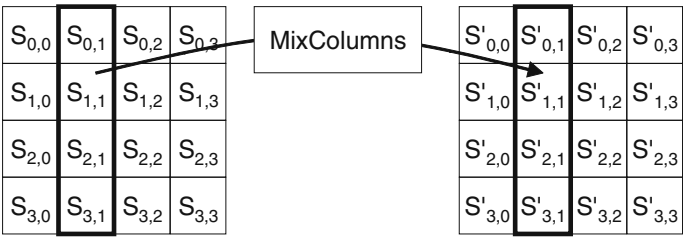

Fig. 10.7 *MixColumns* transforming a column of a State.

$R = K[X]/(X^4+1)$. Addition of these polynomials corresponds to bit-wise XOR of their coefficients. Their multiplication is reduced modulo $M(X) = X^4 + 1$.

Since $X^j \bmod (X^4+1) = X^{j \bmod 4}$, the operation consisting of multiplication of $a(X) = a_3X^3 + a_2X^2 + a_1X + a_0$ by a fixed polynomial $c(X) = c_3X^3 + c_2X^2 + c_1X + c_0$ gives a product

$$\begin{aligned} B(X) &= b_3X^3 + b_2X^2 + b_1X + b_0 \\ &= (c_3a_0 + c_2a_1 + c_1a_2 + c_0a_3)X^3 \\ &\quad + (c_2a_0 + c_1a_1 + c_0a_2 + c_3a_3)X^2 \\ &\quad + (c_1a_0 + c_0a_1 + c_3a_2 + c_2a_3)X \\ &\quad (c_0a_0 + c_3a_1 + c_2a_2 + c_1a_3) \end{aligned} \tag{10.8}$$

This operation can be written as a multiplication of a vector $[A]$ by a circular matrix $[C]$:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_3 & c_2 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & c_2 & c_1 & c_0 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.9}$$

The polynomial $M(X)$ was selected such that it effectively shifts rows of the State. From the cryptographical point of view, this operation mixes bytes across the State and creates a strong dependence between all input bytes $a_0 \ldots a_3$ and an output byte b_i.

The *MixColumns* transformation multiplies each column of the State by a constant polynomial $c(X)$ in the ring R. The $c(X)$ is defined as follows:

$$c(X) = (x+1)X^3 + X^2 + X + x \tag{10.10}$$

The *InvMixColumns* transformation is the inverse of the *MixColumns* operation. *InvMixColumns* multiplies each column of the State by

$$d(X) = (x^3 + x + 1)X^3 + (x^3 + x^2 + 1)X^2 + (x^3 + 1)X + (x^3 + x^2 + x) \tag{10.11}$$

where $d(X) = c^{-1}(X)$ is the inverse of $c(X)$ in R. Polynomials $c(X)$ and $d(X)$ are often expressed with coefficients in the hexadecimal format:

$$c(X) = 03 \cdot X^3 + 01 \cdot X^2 + 01 \cdot X + 02 \tag{10.12}$$

$$d(X) = 0B \cdot X^3 + 0D \cdot X^2 + 09 \cdot X + 0E \tag{10.13}$$

Multiplication of one column of the State by $c(X)$ in R (part of the *MixColumns* operation) can be written in a matrix form:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.14}$$

The expression of the *InvMixColumns* operation in a matrix form is as follows:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0E\ 0B\ 0D\ 09 \\ 09\ 0E\ 0B\ 0D \\ 0D\ 09\ 0E\ 0B \\ 0B\ 0D\ 09\ 0E \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.15}$$

Polynomials representing columns of a State have coefficients which are considered as polynomials (of degree of at most 7) with coefficients in Galois field $GF(2)$. A byte $a(x)$ (or a in simplified notation) is a sum $a(x) = \sum_{0 \le i \le 7} \alpha_i x^i$, where $\alpha_i \in \{0,1\}$. In other words, bytes a are elements of the Galois field $K = GF(2^8)$ constructed using the reduction polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

$$K = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1) \tag{10.16}$$

Addition of polynomials in K corresponds to simple bit-wise exclusive OR (XOR) of the polynomial coefficients. Multiplication of polynomials in the field K corresponds to their multiplication modulo irreducible polynomial $m(x)$ from Equation (10.3). The same polynomial is used in the *SubBytes* operation for calculation of a multiplicative inverse.

10.2.3 Iterative Structure

A flowchart describing AES encryption in terms of basic operations—*SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*—is shown in Figure 10.8. Please note

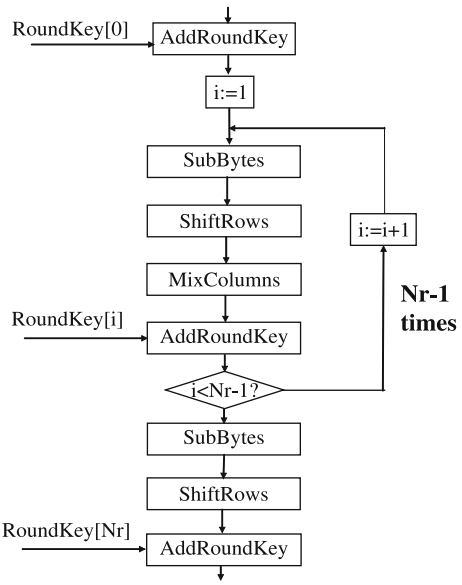

Fig. 10.8 AES encryption flowchart.

that the number of cipher rounds, Nr, depends on the size of an encryption key. The first round is preceded by an initial transformation *AddRoundKey*, in agreement with a generic structure of an iterative block cipher shown in Figure 10.1. The last round, number Nr, is slightly different from the remaining $Nr-1$ rounds, in that it does not contain the *MixColumns* operation.

By a straightforward inversion of the order of operations and by replacing all basic operations by their respective inverses, we obtain the AES decryption flowchart shown in Figure 10.9. Simple regrouping of basic operations leads to an equivalent decryption flowchart, shown in Figure 10.10, which has the same basic structure as an encryption flowchart. The differences amount to providing round keys in the reverse order, replacing all basic operations by their inverses, and swapping the order of operations one and two, and three and four within each round. The operations number one and two during each round of encryption, *SubBytes* and *ShiftRows*, can be performed in an arbitrary order, as shown in Figure 10.11a. Similarly, the operations number one and two during each round of decryption *InvShiftRows* and *InvSubBytes* can be swapped without affecting the result (see Figure 10.11b). By applying this last change, we obtain the decryption flowchart, shown in Figure 10.12, which is most often used as a basis of hardware implementation.

10.2.4 Key Scheduling

Key scheduling in AES is a process aimed at generating $(Nr+1)$ *round keys* based on a single external key. This process consists of two phases called `KeyExpansion`

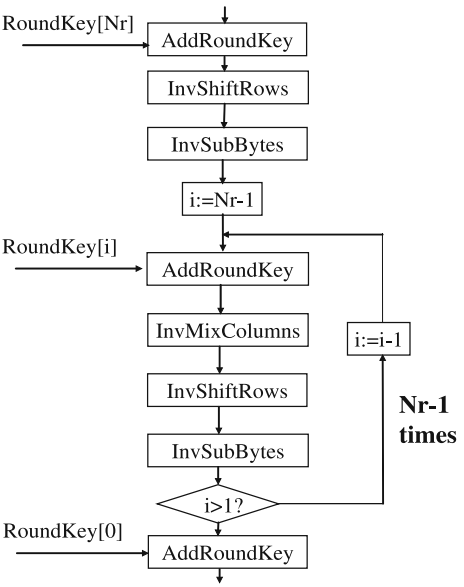

Fig. 10.9 AES decryption flowchart obtained by the straightforward inversion of the encryption flowchart.

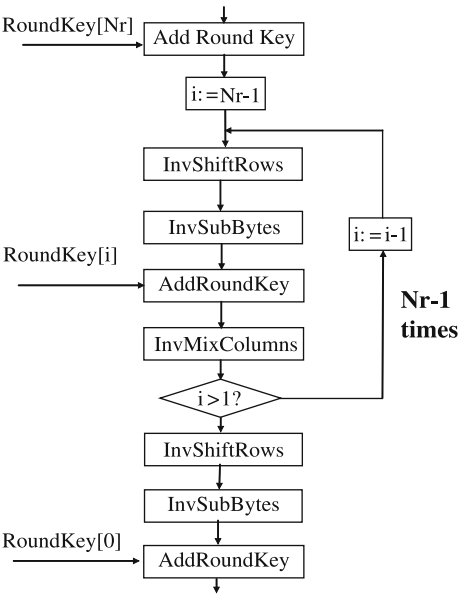

Fig. 10.10 AES decryption flowchart after regrouping of basic operations.

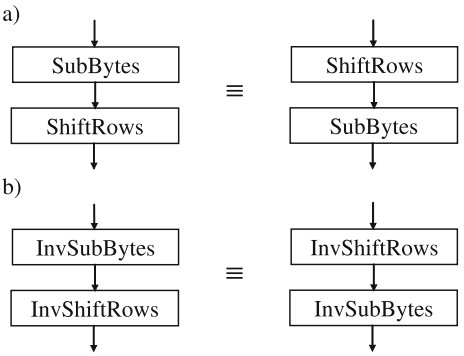

Fig. 10.11 Equivalence between two sequences of basic operations: (**a**) *SubBytes* followed by *ShiftRows*, (**b**) *InvSubBytes* followed by *InvShiftRows*.

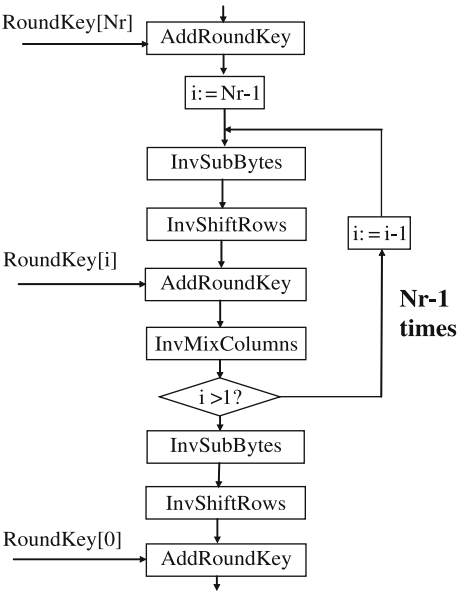

Fig. 10.12 AES decryption flowchart after regrouping of basic operations and swapping *InvSubBytes* with *InvShiftRows*.

and `RoundKeySelection`, as shown in Figure 10.13. Please note that all rectangular fields in this and two subsequent figures correspond to 32-bit words (and not single bytes).

The pseudocode of `KeyExpansion` is shown in Figure 10.16. The output array of words `k[i]` is first initialized with the `Nk` words of the external key, Key. For majority of subsequent values of `i`, `k[i]` is computed by simply XORing an immediately preceding word `k[i-1]` with a word `Nk` positions earlier `k[i-Nk]`, as shown in Figure 10.14. If an index `i` is a multiple of `Nk`, a different transfor-

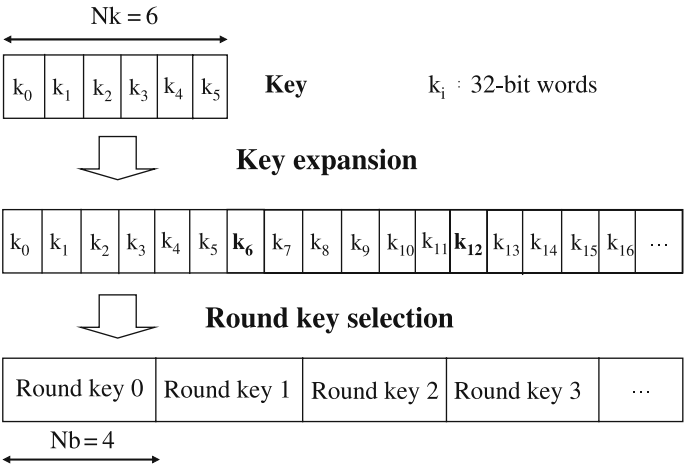
Nk = 6
k0 k1 k2 k3 k4 k5
Key
ki : 32-bit words
Key expansion
k0 k1 k2 k3 k4 k5 k6 k7 k8 k9 k10 k11 k12 k13 k14 k15 k16 ⋯
Round key selection
Round key 0
Round key 1
Round key 2
Round key 3
⋯
Nb = 4

Fig. 10.13 Decomposition of key scheduling into *KeyExpansion* and *RoundKeySelection* for the case of Nk = 6 (192-bit key) and Nb = 4 (128-bit data block).

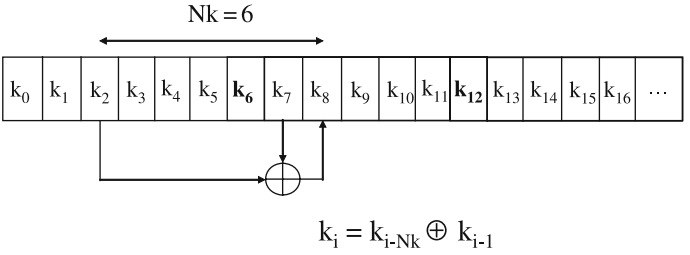

Fig. 10.14 Formula for *KeyExpansion* for i mod $Nk \neq 0$.

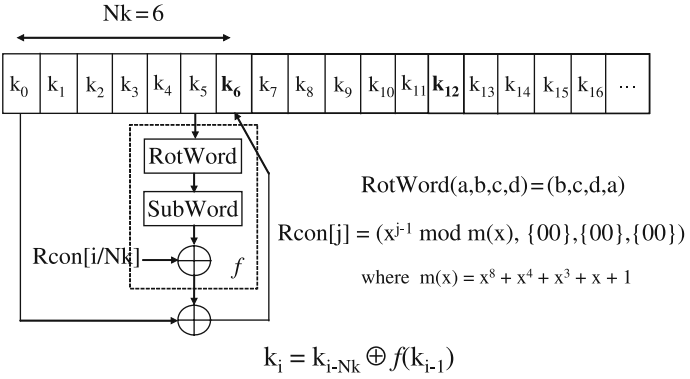
Nk = 6
k0 k1 k2 k3 k4 k5 k6 k7 k8 k9 k10 k11 k12 k13 k14 k15 k16 ⋯
RotWord
SubWord
Rcon[i/Nk]
f
RotWord(a,b,c,d)=(b,c,d,a)
Rcon[j] = (x^j-1 mod m(x), {00},{00},{00})
where m(x) = x^8 + x^4 + x^3 + x + 1
ki = ki-Nk ⊕ f(ki-1)

Fig. 10.15 Formula for *KeyExpansion* for i mod $Nk = 0$.

```
KeyExpansion(byte Key[4*Nk], word k[Nb*(Nr+1)], Nk)
begin
  word temp

  i=0
  while (i < Nk)
    k[i] = word(Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3])
    i = i+1
  end while

  i = Nk

  while (i < Nb * (Nr+1))
    temp = k[i-1]
    if (i mod Nk = 0)
      temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
    else if ((Nk > 6) and (i mod Nk = 4))
      temp = SubWord(temp)
    end if
    k[i] = k[i-Nk] xor temp
    i = i + 1
  end while

end
```

Fig. 10.16 Pseudocode for the *KeyExpansion* phase of *KeyScheduling*.

mation, shown in Figure 10.15, is used. In this transformation, *RotWord* is a cyclic rotation of bytes within a word, *SubWord* is a *SubBytes* transformation applied independently to each byte of an input word, and `Rcon[i]` is an array of four constants defined in $GF(2^8)$. If `(Nk > 6) and (i mod Nk) = 4`, a simplified version of the same transformation is applied.

10.3 FPGA and ASIC Technologies

Cryptographic transformations can be implemented in both software and hardware. Software implementations are designed and coded in programming languages, such as C, C++, Java, and assembly language, to be executed, among others, on general purpose microprocessors, digital signal processors, and smart cards. Hardware implementations are designed and coded in hardware description languages, such as VHDL and Verilog HDL, and are intended to be realized using two major implementation approaches: application-specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs).

Application-specific integrated circuits (ASICs) are designed all the way from the behavioral description to the physical layout and then sent for a fabrication in a semiconductor foundry. Field programmable gate array (FPGA) can be bought off the shelf and reconfigured by designers themselves. With each reconfiguration,

which takes only a fraction of a second, an integrated circuit can perform a completely different function.

FPGA consists of thousands of universal reconfigurable logic blocks, connected using reconfigurable interconnects and switches, as shown in Figure 10.17. Additionally, modern FPGAs contain embedded higher-level components, such as memory blocks, multipliers, multipliers–accumulators, and even microprocessor cores. Reconfigurable input/output blocks provide a flexible interface with the outside world. Reconfiguration, which typically lasts only a fraction of a second, can change a function of each building block and interconnects among them, leading to a functionally new digital circuit.

In Table 10.1, we collect and contrast features of implementations of cryptographic transformations based on ASICs and FPGAs (hardware) and microprocessors (software). The performance characteristics of ASICs and FPGAs are almost identical, as demonstrated by the first group of features, and substantially different from the performance characteristics of general purpose microprocessors. Both ASICs and FPGAs can make a full use of parallel processing and pipelining and operate on arbitrary size words. In general purpose microprocessors, parallel processing and pipelining are limited by the number and internal structure of the processor functional units and by the instruction level parallelism. Additionally, all functional units operate on the fixed-size arguments only.

The primary difference between ASICs and FPGAs in terms of the performance characteristics is a smaller speed of FPGAs caused by the delays introduced by the circuitry required for reconfiguration. As a result of this speed penalty, any digital circuit implemented in an FPGA is typically slower than the same circuit implemented in an ASIC, assuming that both integrated circuits are fabricated using the same semiconductor technology (in particular, using the same transistor size).

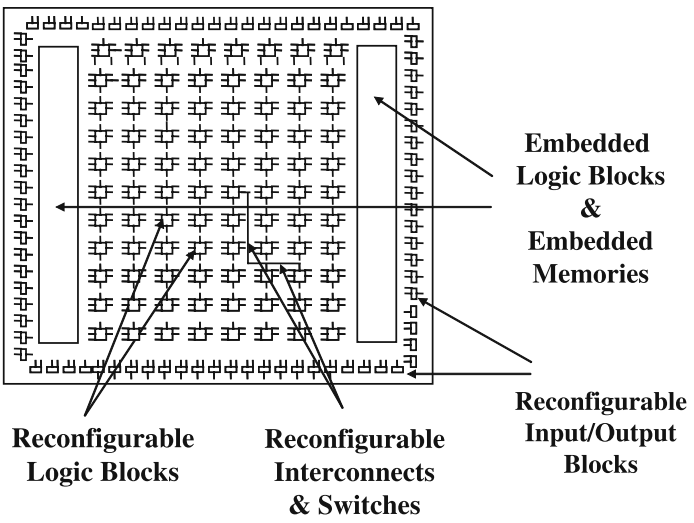

Fig. 10.17 General structure and main building blocks of an FPGA.

Table 10.1 Characteristic features of implementations of cryptographic transformations in ASICs, FPGAs, and microprocessors.

	ASICs	FPGAs	Microprocessors
Performance characteristics			
Parallel processing	Yes	Yes	Limited
Pipelining	Yes	Yes	Limited
Word size	Variable	Variable	Fixed
Speed	Very fast	Fast	Moderately fast
Functionality			
Algorithm agility	No	Yes	Yes
Tamper resistance	Strong	Limited	Weak
Access control to keys	Strong	Moderate	Weak
Development process			
Description languages	VHDL, Verilog HDL	VHDL, Verilog HDL	C, C++, Java, assembly language
Design cycle	Long	Moderately long	Short
Design tools	Very expensive	Moderately expensive	Inexpensive
Maintenance and upgrades	Expensive	Inexpensive	Inexpensive

The recent study performed at the University of Toronto [24] quantified the performance differences between the current generation of ASICs and FPGAs. A set of 23 benchmarks covering applications in the area of cryptography, digital signal processing, and communications were included in the study. The study concluded that for circuits containing only combinational logic and flip-flops, the ratio of silicon area required to implement them in FPGAs and ASICs is on average 40. For circuits that could take advantage of dedicated blocks present in modern FPGAs, such as multiplier/accumulators and block memories, these blocks reduced the average area gap significantly to as little as 21. The ratio of critical path delay, from FPGA to ASIC, was found to be roughly 3–4, with less influence from embedded memories and embedded logic blocks. The dynamic power consumption ratio was approximately 12 times and, with hard blocks, this gap generally became smaller.

The common features of FPGAs and microprocessors concern mostly functionality and do not affect performance. Both general purpose microprocessors and FPGAs can be easily reconfigured in real time to perform a different algorithm. The disadvantage of this feature is a limited tamper resistance; the contents of an FPGA can be, at least in theory, modified by an unauthorized user. In practice, the contents of an FPGA are typically downloaded during the initialization from the read-only memory, such as EPROM, which cannot be easily tampered with, at least remotely. The access control to cryptographic keys in FPGAs is also stronger than in software, but weaker than in ASICs.

The development process in both hardware implementation approaches is very similar. In both FPGAs and ASICs, the circuit is described using a hardware description language, verified using a digital circuit simulator, and then tested using a prototyping board. The primary difference between FPGAs and ASICs is that FPGAs do not require the physical design (layout), fabrication, and testing for

physical defects. As a result, the design cycle is significantly shorter and the design tools and testing much less expensive. The interesting similarity between FPGAs and software is a possibility of remote maintenance and upgrading, based on electronic patches.

10.4 Parameters of Hardware Implementations

Hardware implementations of secret-key ciphers can be characterized using several performance parameters. Below we provide our definitions of major parameters and derive formulas that demonstrate mutual dependencies among these parameters.

10.4.1 Throughput and Latency

Encryption (decryption) throughput is defined as the number of bits encrypted (decrypted) in a unit of time. Typically, the encryption and decryption throughputs are equal, and therefore only one parameter is reported. A typical unit of throughput is Mbit/s (megabit per second) or Gbit/s (gigabit per second). It is worth mentioning that 1 Mbit/s = 10^6 bit/s, and not 2^{20} bit/s, and 1 Gbit/s = 10^9 bit/s, and not 2^{30} bit/s.

Encryption (decryption) latency is defined as the time necessary to encrypt (decrypt) a single block of plaintext (ciphertext). The typical unit of latency in the current technology is ns (nanosecond).

The encryption (decryption) latency and throughput are related by

$$Throughput = \frac{block\_size \cdot number\_of\_blocks\_processed\_simultaneously}{latency} \quad (10.17)$$

In applications where large amounts of data are encrypted or decrypted, throughput determines the total encryption/decryption time and thus is the best measure of the cipher speed. In applications where a small number of plaintext (ciphertext) blocks is processed, the total encryption/decryption time depends on both throughput and latency.

10.4.2 Area

The area required for the cipher implementation is an important parameter for the following reasons:

- Cost
  The area of an integrated circuit is a primary factor determining its cost. It is traditionally assumed that the cost of an integrated circuit is directly proportional

to the circuit area. This dependence is not always accurate, especially taking into account the cost of a package, which is determined by the number of the circuit inputs and outputs.

- Limit on the maximum area
  In certain hardware environments, there exists a limit on the maximum area of a cryptographic unit. This limit may be imposed by the cost, available fabrication technology, power consumption, or any combination of these factors. For example, in smart cards and microcontrollers, both cost and power consumption limit the area of the embedded encryption units; in FPGAs, the area is limited by the available fabrication technology and the cost of a programmable device.

In ASIC implementations, the area required by the cryptographic unit is typically expressed in μm^2. Two related measures are the transistor count and the logic gate count. Values of all three measures are closely correlated, but not necessarily strictly proportional to each other. All three measures are reported by the tools used for the automated logic synthesis of ASICs. In the semi-custom design methodology, these values are a function of the standard cell library used during logic synthesis.

In FPGA implementations, the only circuit size measures reported by the CAD tools are the number of basic configurable logic blocks and the number of equivalent logic gates. It is commonly believed that out of these two measures, the number of basic configurable logic blocks approximates the circuit area more accurately. Measuring and comparing circuit area in FPGAs are additionally complicated by the existence of embedded logic blocks and embedded memories. The specifications of FPGA devices typically do not provide any information about the relative ratio of the areas used by embedded blocks and basic reconfigurable logic blocks.

10.5 Hardware Architectures of Symmetric Block Ciphers

10.5.1 Hardware Architectures vs. Block Cipher Modes of Operation

Symmetric-key block ciphers are used in several operating modes. From the point of view of hardware implementations, these modes can be divided into two major categories:

1. Non-feedback modes, such as electronic code book mode (ECB) and counter mode (CTR).
2. Feedback modes, such as cipher block chaining mode (CBC), cipher feedback mode (CFB), and output feedback mode (OFB).

In the non-feedback modes, encryption of each subsequent block of data can be performed independently from processing other blocks. In particular, all blocks can be encrypted in parallel. In the feedback modes, it is not possible to start encrypting the next block of data until encryption of the previous block is completed.

As a result, all blocks must be encrypted sequentially, with no capability for parallel processing. The limitation imposed by the feedback modes does not concern decryption, which can be performed on several blocks of ciphertext in parallel for both feedback and non-feedback operating modes.

In the old security standards, the encryption of data was performed primarily using feedback modes, such as CBC and CFB. Using these standards did not permit to fully utilize the performance advantage of the hardware implementations of secret-key ciphers, based on parallel processing of multiple blocks of data. The situation has been partially remedied by including a counter mode in the NIST recommendations on the AES modes of operation. Other non-feedback modes of operation are currently under investigation by the cryptographic community.

10.5.2 Basic Iterative Architecture

The basic hardware architecture used to implement an encryption/decryption unit of a typical secret-key cipher is shown in Figure 10.18. One round of the cipher is implemented as a combinational logic and supplemented with a single register and a multiplexer. In the first clock cycle, input block of data is fed to the circuit through the multiplexer and stored in the register. In each subsequent clock cycle, one round of the cipher is evaluated, the result is fed back to the circuit through the multiplexer, and stored in the register. The two characteristic features of this architecture are

- Only one block of data is encrypted at a time.
- The number of clock cycles necessary to encrypt a single block of data is equal to the number of cipher rounds, #rounds.

The throughput and latency of the basic iterative architecture, $Throughput_{iterative}$ and $Latency_{iterative}$, are given by

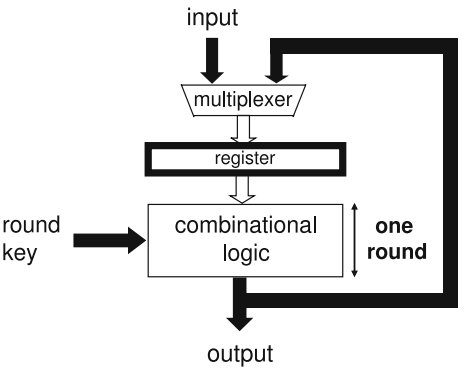

Fig. 10.18 Basic iterative architecture of a block cipher.

$$Throughput_{iterative} = \frac{block\_size}{\#rounds \cdot T_{CLK_{iterative}}} \tag{10.18}$$

$$Latency_{iterative} = \#rounds \cdot T_{CLK_{iterative}} \tag{10.19}$$

where $T_{CLK_{iterative}}$ is a clock period of the basic iterative architecture.

10.5.3 Loop Unrolling

An architecture *with partial loop unrolling* is shown in Figure 10.19b. The only difference compared to the basic iterative architecture is that the combinational part of the circuit implements K rounds of the cipher, instead of a single round. K must be a divisor of the total number of rounds, #*rounds*.

The number of clock cycles necessary to encrypt a single block of data decreases by a factor of K. At the same time the minimum clock period increases by a factor slightly smaller than K, leading to an overall relatively small increase in the encryption throughput, and decrease in the encryption latency, as shown in Figure 10.20. Because the combinational part of the circuit constitutes the majority of the circuit area, the total area of the encryption/decryption unit increases almost proportionally to the number of unrolled rounds, K. Additionally, the number of internal keys used in a single clock cycle increases by a factor of K, which in hardware implementations typically implies the almost proportional growth in the area used to store internal keys.

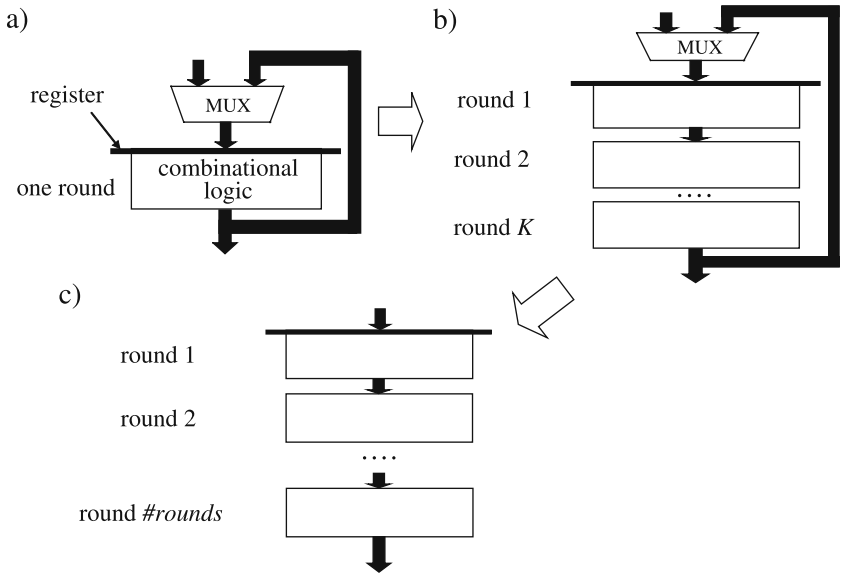

Fig. 10.19 Three hardware architectures suitable for feedback cipher modes: (**a**) basic iterative, (**b**) with partial loop unrolling, (**c**) with full loop unrolling.

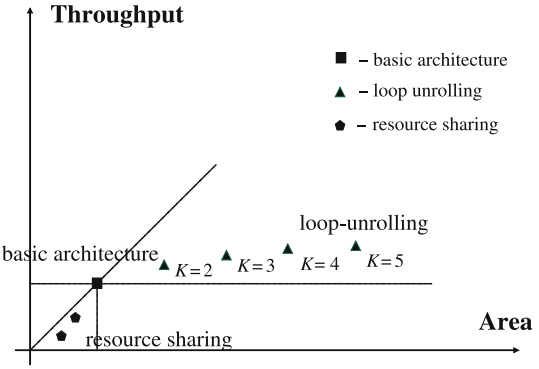

Fig. 10.20 Throughput vs. area characteristics of hardware architectures suitable for feedback cipher modes.

Architecture *with full loop unrolling* is shown in Figure 10.19c. The input multiplexer and the feedback loop are no longer necessary, leading to a small increase in the cipher speed and decrease in the circuit area compared to the partial loop unrolling with the same number of rounds unrolled.

In summary, loop unrolling enables increasing the circuit speed in both feedback and non-feedback operating modes. Nevertheless this increase is relatively small and incurs a large area penalty. As a result, choosing this architecture can be justified only for feedback cipher modes, where none other architecture offers speed greater than the basic iterative architecture, and only for implementations where large increase in the circuit area can be tolerated.

10.5.4 Pipelining

A traditional methodology for design of high-performance implementations of secret-key block ciphers operating in non-feedback cipher modes is shown in Figure 10.21. The basic iterative architecture, shown in Figure 10.21a, is implemented first and its speed and area determined. Based on these estimations, the number of rounds K that can be unrolled without exceeding the available circuit area is found. The number of unrolled rounds, K, must be a divisor of the total number of cipher rounds, #rounds. If the available circuit area is not large enough to fit all cipher rounds, architecture with partial outer-round pipelining, shown in Figure 10.21b, is applied. The difference between this architecture and the architecture with partial loop unrolling, shown in Figure 10.19b, is the presence of registers inside of the combinational logic on the boundaries between any two subsequent cipher rounds. As a result, K blocks of data can be processed by the circuit at the same time, with each of these blocks stored in a different register at the end of a clock cycle. This technique of parallel processing of multiple streams of data by the same circuit is called pipelining. The throughput and area of the circuit with partial outer-round

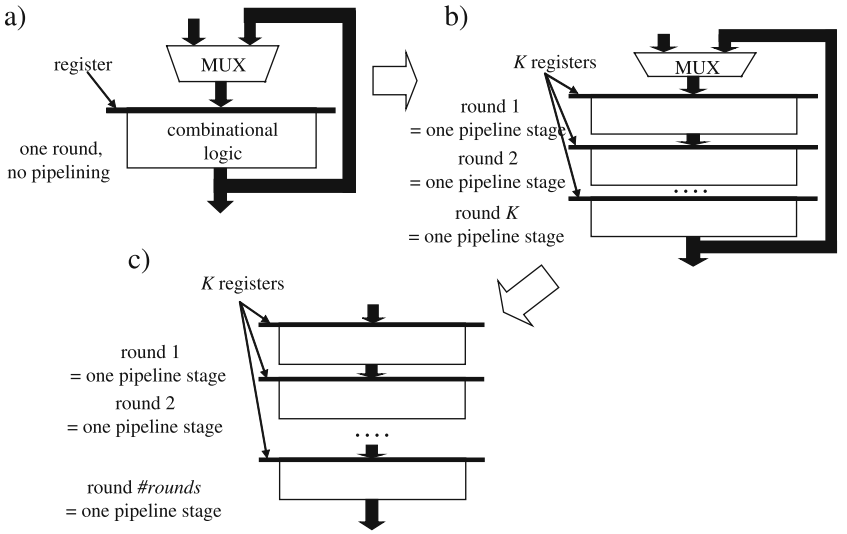

Fig. 10.21 Three hardware architectures used traditionally to implement non-feedback cipher modes: (**a**) basic iterative, (**b**) with partial outer-round pipelining, (**c**) with full outer-round pipelining.

pipelining increase proportionally to the value of K, as shown in Figure 10.23, the encryption/decryption latency remains the same as in the basic iterative architecture, as shown in Figure 10.24. If the available area is large enough to fit all cipher rounds, the feedback loop is no longer necessary and full outer-round pipelining, shown in Figure 10.21c, can be applied.

An optimized design methodology for implementing non-feedback cipher modes is shown in Figure 10.22. Before loop unrolling, the optimum number of pipeline registers is inserted inside of a cipher round, as shown in Figure 10.22b. The entire round, including internal pipeline registers is then repeated K times (see Figure 10.22c). The number of unrolled rounds K depends on the maximum available area or the maximum required throughput.

The primary advantage of the latter methodology is shown in Figure 10.23. Inserting registers inside of a cipher round significantly increases cipher throughput at the cost of only marginal increase in the circuit area. As a result, the throughput to area ratio increases until the number of internal pipeline stages reaches its optimum value k_{opt}. Inserting additional registers may still increase the circuit throughput, but the throughput to area ratio will deteriorate. The throughput to area ratio remains unchanged during the subsequent loop unrolling. The throughput of the circuit is given by

$$Throughput_{pipelined}(K,k) = \frac{K \cdot block\_size}{\#rounds \cdot T_{CLK_{inner\_round}}(k)} \tag{10.20}$$

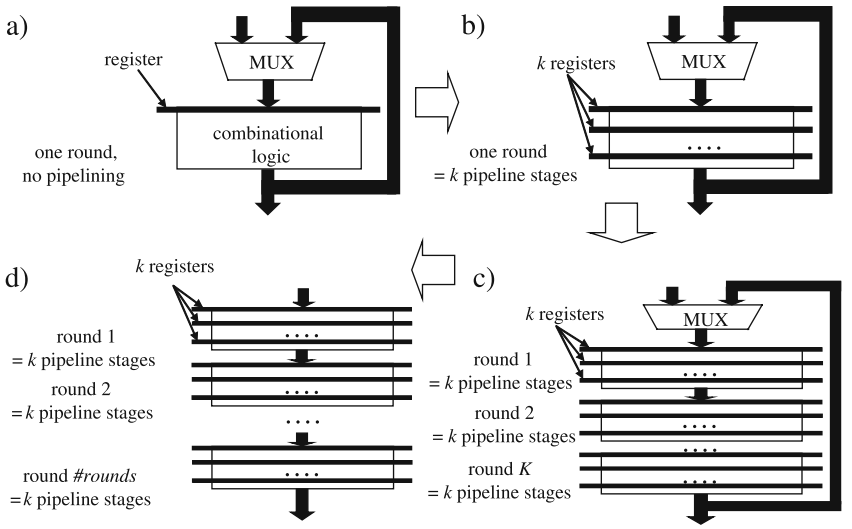

Fig. 10.22 Four hardware architectures suitable for non-feedback cipher modes: (**a**) basic iterative, (**b**) with inner-round pipelining, (**c**) with partial inner- and outer-round pipelining, (**d**) with full inner- and outer-round pipelining.

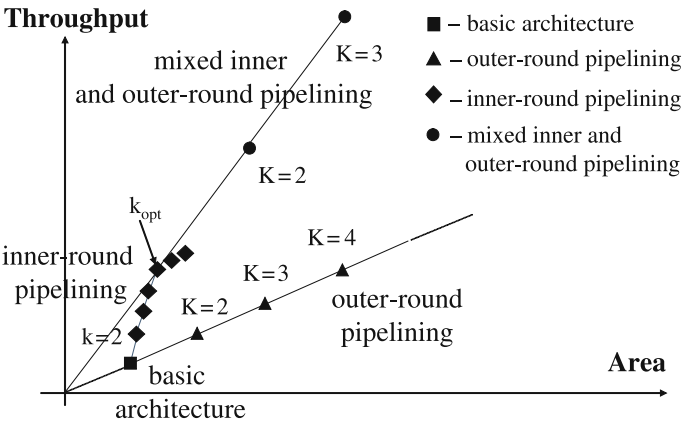

Fig. 10.23 Throughput vs. area characteristics of hardware architectures suitable for non-feedback cipher modes.

where k is the number of inner-round pipeline stages, K is the number of outer-round pipeline stages, and $T_{CLK_{inner\_round}}(k)$ is the clock period in the architecture with the k-stage inner-round pipelining. For a given limit in the circuit area, mixed inner- and outer-round pipelining shown in Figure 10.22c offers significantly higher throughput compared to the pure outer-round pipelining (see Figure 10.23).

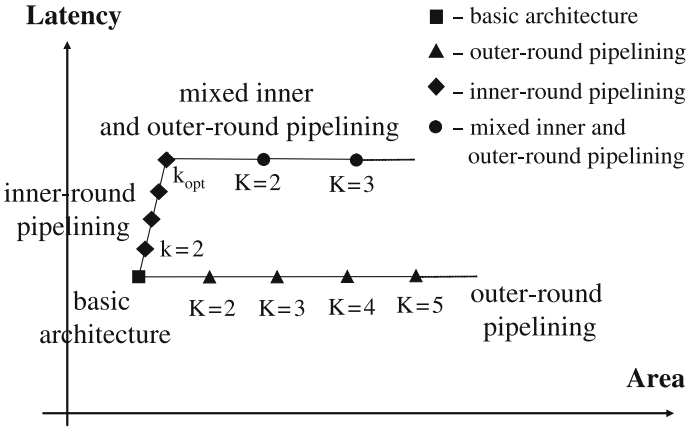

Fig. 10.24 Latency vs. area characteristics of hardware architectures suitable for non-feedback cipher modes.

When the limit on the circuit area is large enough, all rounds of the cipher can be unrolled, as shown in Figure 10.22d, leading to the throughput given by

$$Throughput_{fully\_pipelined}(K, k_{opt}) = \frac{block\_size}{T_{CLK_{inner\_round}}(k_{opt})} \tag{10.21}$$

where k_{opt} is the number of inner-round pipeline stages optimum from the point of view of the throughput to area ratio. The only side effect of our methodology is the increase in the encryption/decryption latency. This latency is given by

$$Latency_{fully\_pipelined}(K, k) = \#rounds \cdot k \cdot T_{CLK_{inner\_round}}(k) \tag{10.22}$$

This latency does not depend on the number of rounds unrolled, K. The increase in the encryption/decryption latency, typically in the range of single microseconds, usually does not have any major influence on the operation of the high-volume cryptographic system optimized for maximum throughput. This is particularly true for applications with a human operator present on at least one end of the secure communication channel.

The input/output timing characteristics of three basic secret-key cipher architectures are shown in Figure 10.25. In the basic iterative architecture, a new block of data must be fed into the system only once per *#rounds* clock cycles. In case of the inner-round pipelining, there are periods of time when the input must be fed into the cryptographic core every clock cycle, even though an average input/output throughput is much lower (Figure 10.25b). In the full mixed inner- and outer-round pipelining, input blocks are fed to the encryption unit every clock cycle (Figure 10.25c).

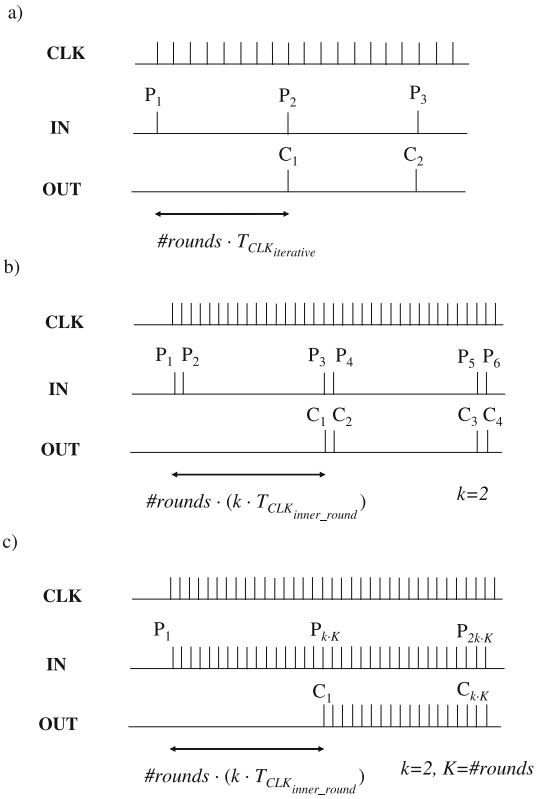

Fig. 10.25 Input/output timing characteristics of various architectures: (**a**) basic iterative architecture, (**b**) inner-round pipelining, (**c**) full inner- and outer-round pipelining.

10.5.5 Limits on the Maximum Clock Frequency of Pipelined Architectures

Throughput of the architecture with the mixed inner- and outer-round pipelining is directly proportional to the maximum clock frequency for the inner-round pipelining (see Equation (10.20)). The following factors may limit the maximum clock frequency,

$$f_{CLK_{inner\_round}}(k) = \frac{1}{T_{CLK_{inner\_round}}(k)} \tag{10.23}$$

in this architecture:

1. *delay of a single round divided by k*
   For small values of k, it is usually possible to divide the combinational portion of a single round into k stages with equal (or at least approximately equal) delays. The delay of a single stage, equal to the delay of a single round divided by k,

determines the minimum clock period of the circuit, $T_{CLK_{inner\_round}}(k)$, as shown in Figure 10.26a.

2. *delay of the largest indivisible operation*
   For some ciphers, when the number of internal pipeline stages k increases, it becomes more and more difficult to divide the combinational portion of a single round into stages with equal delays. At certain point, introducing additional internal registers to the circuit may require dividing an elementary operation of the cipher, such as an S-box or addition, into several stages. This division may be difficult to accomplish if the operation is performed using a standard library cell, look-up table, special carry propagate circuitry, or if the operation is so simple that it cannot be easily divided into less-complex atomic operations. This case is shown in Figure 10.26b.
3. *delay of the control unit*
   The control unit determines the data flow in the circuit. This unit is responsible for generating enable signals for all registers and memories in the circuit and address inputs for all memories and major multiplexers. The time necessary to generate and distribute these signals, counted from the rising edge of the clock, may be greater than the time necessary to propagate data between two adjacent registers in the pipeline, as shown in Figure 10.26c. This is especially true

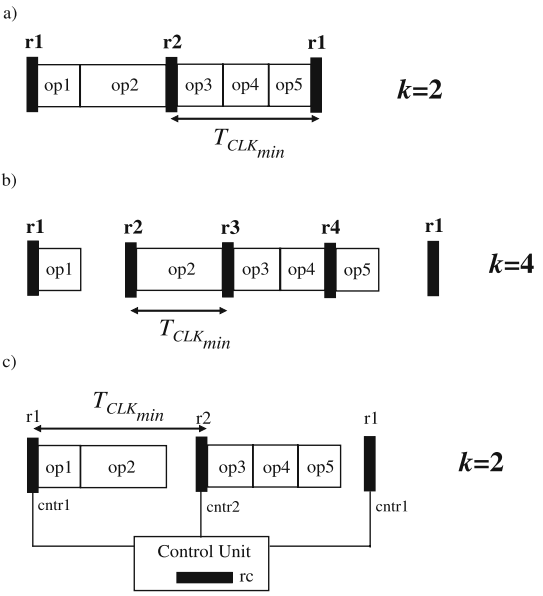

Fig. 10.26 Limits on the minimum clock period in the architecture with inner-round pipelining: (**a**) ideal situation, evenly divided round; (**b**) clock period limited by the largest indivisible operation; (**c**) clock period limited by the control unit, i.e., the time necessary to generate and distribute control signals.

for control signals with large fanouts distributed globally to every stage of the pipeline.

4. *limit on the maximum latency*
   Increasing the number of inner-round pipeline stages, k, increases the overall latency of the cipher, by a factor of approximately $(k-1)\cdot(t_P+t_{su})$, where t_P and t_{su} denote the propagation delay and the setup time of a register, respectively. This approximation does not take into account any changes in the routing (interconnect) delays. If the specification of the cryptographic system imposes a limit on the maximum latency, $Latency_{max}$, this limit may determine the maximum possible number of inner-round pipeline stages, k_{max}.

$$k_{max} \leq \frac{(Latency_{max} - Latency_{iterative})}{\#rounds \cdot (t_P + t_{su})} \tag{10.24}$$

5. *limit on the maximum input/output bandwidth*
   We define the input/output bandwidth as a frequency of an external clock used to control the transmission of data between the integrated circuit and an external environment. The input/output bandwidth necessary to sustain the throughput of the circuit working in the mixed inner- and outer-round pipelining is given by

$$Bandwidth = \frac{Throughput(K,k)}{bus\_width} = \frac{K}{\#rounds} \cdot \frac{block\_size}{bus\_width} \cdot f_{CLK_{inner\_round}}(k) \tag{10.25}$$

   where $f_{CLK_{innerround}}(k)$ is a frequency of the clock for a k-round inner-round pipelining. The circuit is assumed to have two independent ports of the width bus_width used for input and output, respectively. In case of using the same bus for both input and output, the bandwidth must be at least twice as high to sustain the same throughput. The maximum bandwidth may limit the maximum value of the product $K \cdot f_{CLK_{inner\_round}}(k)$ and thus the maximum number of inner- and outer-round pipeline stages.

10.5.6 Compact Architectures with Resource Sharing

For majority of ciphers, it is possible to significantly decrease the circuit area by time sharing of certain resources (e.g., S-boxes in AES). This is accomplished by using the same functional unit to process two (or more) parts of the data block in different clock cycles, as shown in Figure 10.27.

In Figure 10.27a, two parts of the data block, $D0$ and $D1$, are processed in parallel, using two independent functional units F. In Figure 10.27b, a single unit F is used to process two parts of the data block sequentially, during two subsequent

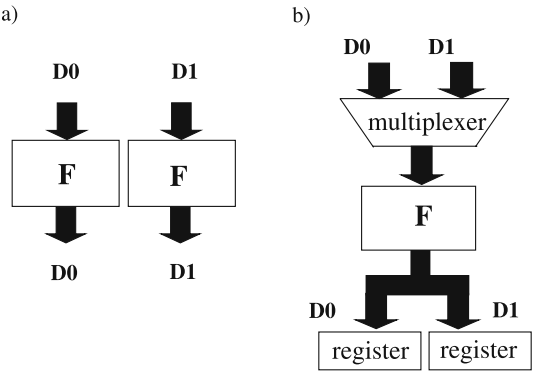

Fig. 10.27 Basic idea of resource sharing: (**a**) parallel execution of two instantiations of the functional unit F, no resource sharing; (**b**) resource sharing of the functional unit F.

clock cycles. This technique can be used to reduce the basic datapath in AES from 128 bits to 64 bits, 32 bits, and even 8 bits.

10.6 Implementation of Basic Operations of AES in Hardware

10.6.1 SubBytes and InvSubBytes

10.6.1.1 Look-Up Table

SubBytes is composed of 16 identical 8×8 S-boxes working in parallel. *InvSubBytes* is composed of the same number of 8×8-bit inverse S-boxes. Each of these S-boxes can be implemented independently using a 256×8-bit look-up table. A look-up table is implemented in digital systems using ROM (read-only memory). In this memory, input to an S-box is connected to the address lines, and the output is obtained at the data out bus.

Each of the AES *SubBytes* look-up tables is of the size of 256 bytes = 2048 bits = 2 kilobits. If encryption and decryption are implemented together within the same circuit, both uninverted and inverted 256 byte look-up tables can be placed within one 512 byte memory block. In this case, the most significant bit of an address is a control bit that distinguishes between encryption and decryption.

If a dual port ROM memory is available, which is often the case in FPGAs, the same memory can implement two S-boxes working in parallel.

In Xilinx FPGAs embedded memories are typically implemented as memories with synchronous output. This feature means that the output of the memory does not change until the next rising edge of the clock. This kind of synchronous ROM (read-only memory) is equivalent to a regular asynchronous ROM followed by a register. This feature of Xilinx Block RAMs determines uniquely the location of a register in the basic iterative architecture of AES.

10.6.1.2 Look-Up Table and Logic

The total size of the look-up tables necessary to implement both encryption and decryption can be reduced by a factor of two using knowledge of an internal structure of *SubBytes* and *InvSubBytes*, shown in Figure 10.5. In this case, only *inversion in* $GF(2^8)$ is implemented using look-up tables. These look-up tables are shared between the encryption and decryption units. The *affine transformation* and the *inverse affine transformation* can be implemented easily using an array of XOR gates. Up to 6-input (4-input) XOR gates are required in order to implement affine (inverse affine) transformation using one level of gates. If only 2-input XOR gates are available, up to three (two) layers of such gates might be necessary to implement the same transformations.

10.6.1.3 Logic Only

The amount of memory required to implement *SubBytes* and *InvSubBytes* can be reduced to zero by utilizing the internal logic structure of *inversion in* $GF(2^8)$. This approach makes particular sense for ASIC implementations, in which memory is typically costly in terms of the circuit area.

In FPGAs, memory blocks are always present independently whether they are used or not, but their replacement by logic may be still justified. For example, memory might be already used to implement some other functions, such as input/output buffers. Additionally, in case of deeply pipelined architectures (see Section 10.5.4), memory-based implementation can impose an artificial restriction on the minimum clock period (as described in Section 10.5.5), while the logic-based implementation can be further pipelined.

The basic idea of the logic-only implementation is to notice that inversion in $GF(2^8)$ can be decomposed into a sequence of operations in $GF(2^4)$ (including addition, multiplication, and inversion), as shown in Figure 10.29. Similarly, operations in $GF(2^4)$ can be expressed in terms of operations in $GF(2^2)$ (see Figures 10.30, 10.32, and 10.35) and operations in $GF(2^2)$ in terms of operations in $GF(2)$ (see Figures 10.33, 10.34, 10.36, and 10.37). The operations in $GF(2)$ can be implemented using simple XOR gate (addition) and AND gate (multiplication). An inverse of 1 in $GF(2)$ is 1, and the inverse of 0 does not exist. Thus, the entire *inversion in* $GF(2^8)$ can be decomposed into a logic circuit composed of XOR and AND gates only.

The complexity (number of equivalent logic gates) and critical path (delay) of this circuit depend on the choice of the specific representation for each field $GF(2^{2k})$ using components of the underlying field $GF(2^k)$, for $k = 4, 2$, and 1. The initial choice of the specific representations was provided by Satoh et al. [33]. This choice was later examined by Mentens et al. [28]. The authors compared 64 different polynomial basis representations, and found a representation giving a 5% improvement over [33].

Canright [3, 4] has extended this comparison to include normal basis representations of the components of the fields $GF(2^{2k})$. He investigated a total of 432 different representation choices and concluded that his best choice gives a 20% improvement in terms of the total gate count compared to [33].

The details of the optimum design are shown in the hierarchical form in Figures 10.28, 10.29, 10.30, 10.31, 10.32, 10.33, 10.34, 10.35, 10.36 and 10.37. The top level design of the *SubBytes*/*InvSubBytes* circuit is shown in Figure 10.28.

X is an 8×8 basis conversion matrix, which changes the Galois field representation from the optimum representation used for internal computations within the $GF(2^8)$ inverter to the standard AES polynomial representation used in the remaining calculations. X^{-1} is a matrix describing the conversion in the opposite direction. M is an 8×8 matrix and $b = \{63\}$ is an 8×1 bit vector, where $y' = M \cdot y + b$ is an equation describing the affine transformation of *SubBytes*.

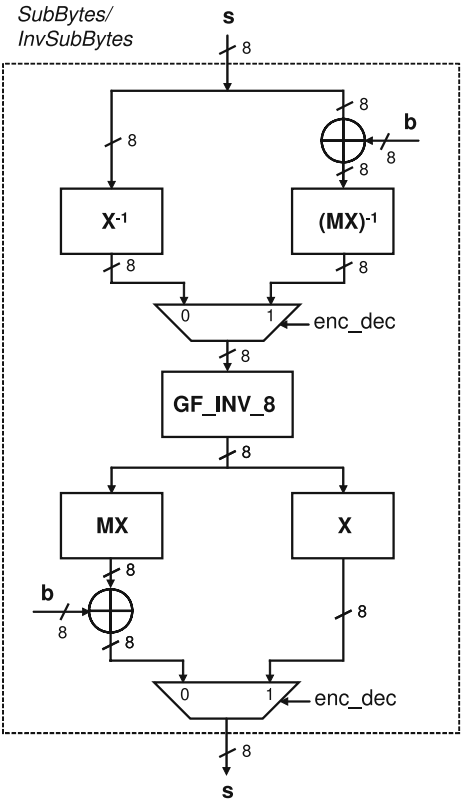

Fig. 10.28 Implementation of *SubBytes* and *InvSubBytes* using logic only, according to [3, 4]. The notation follows conventions introduced in [3]. *enc_dec* is a select signal equal to 0 for encryption and 1 for decryption. X is an 8×8 basis conversion matrix, M is an 8×8 matrix, and b is an 8×1 bit vector, where $y' = M \cdot y + b$, with $b = \{63\}$, is an equation describing the affine transformation of *SubBytes*.

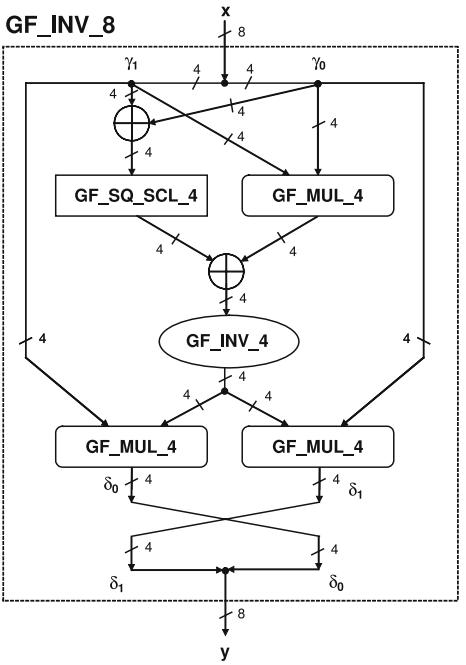

Fig. 10.29 $GF(2^8)$ inverter, GF_INV_8. Notation: GF_INV_4—$GF(2^4)$ inverter (see Figure 10.30), GF_MUL_4—$GF(2^4)$ multiplier (see Figure 10.32), and GF_SQ_SCL_4—$GF(2^4)$ squarer and scaler (see Figure 10.35).

The data path for encryption (see Figure 10.28) includes conversion from the standard polynomial representation to the internal representation, X^{-1}, inversion in $GF(2^8)$, conversion back to the standard representation, X, combined with the multiplication by matrix M of affine transformation, MX, followed by the addition of the vector b of the affine transformation. Thus, the entire *SubBytes* transformation can be described by the equation

$$s' = (MX) \cdot (X^{-1}s)^{-1} + b \tag{10.26}$$

The data path for decryption starts from adding the vector b, followed by the multiplication by M^{-1} and X^{-1}, combined into a product $X^{-1}M^{-1} = (MX)^{-1}$, followed by inversion in $GF(2^8)$, and multiplication by X. Thus, the entire *InvSubBytes* transformation can be described by the equation

$$s' = X \cdot ((MX)^{-1}(s+b))^{-1} \tag{10.27}$$

In both cases, the convention for the order of bits within vectors s' and s is as given below in Equation (10.28).

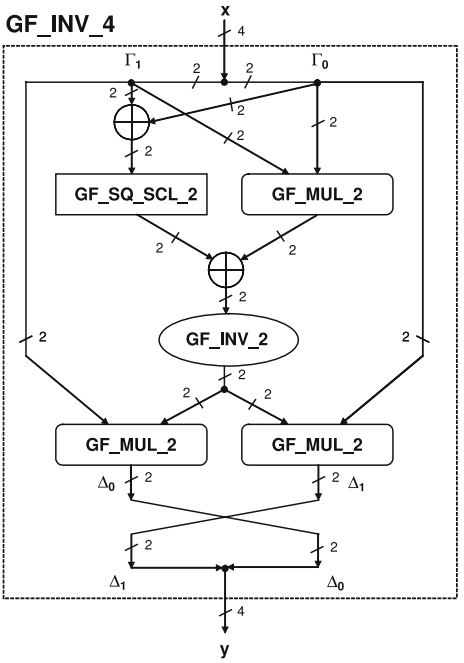

Fig. 10.30 $GF(2^4)$ inverter, GF_INV_4. Notation: GF_INV_2—$GF(2^2)$ inverter (see Figure 10.31), GF_MUL_2—$GF(2^2)$ multiplier (see Figure 10.33), and GF_SQ_SCL_2—$GF(2^2)$ squarer and scaler (see Figure 10.36).

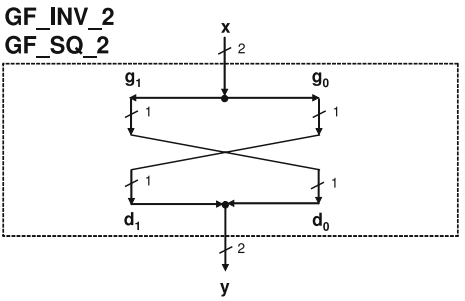

Fig. 10.31 $GF(2^2)$ inverter, GF_INV_2; equivalent to $GF(2^2)$ squarer, GF_SQ_2.

$$s' = \begin{bmatrix} s'_7 \\ s'_6 \\ s'_5 \\ s'_4 \\ s'_3 \\ s'_2 \\ s'_1 \\ s'_0 \end{bmatrix} \qquad s = \begin{bmatrix} s_7 \\ s_6 \\ s_5 \\ s_4 \\ s_3 \\ s_2 \\ s_1 \\ s_0 \end{bmatrix} \tag{10.28}$$

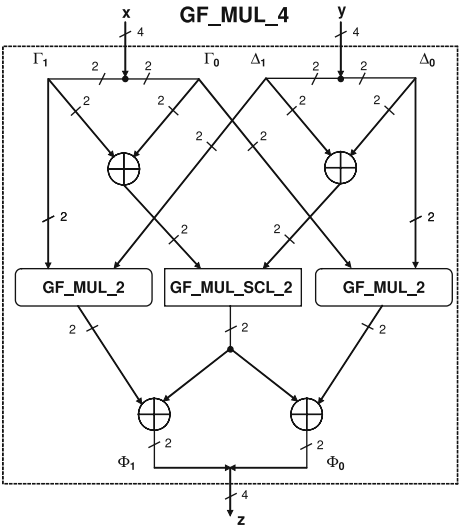

Fig. 10.32 $GF(2^4)$ multiplier, GF_MUL_4. Notation: GF_MUL_2—$GF(2^2)$ multiplier (see Figure 10.33) and GF_MUL_SCL_2—$GF(2^2)$ multiplier and scaler (see Figure 10.34).

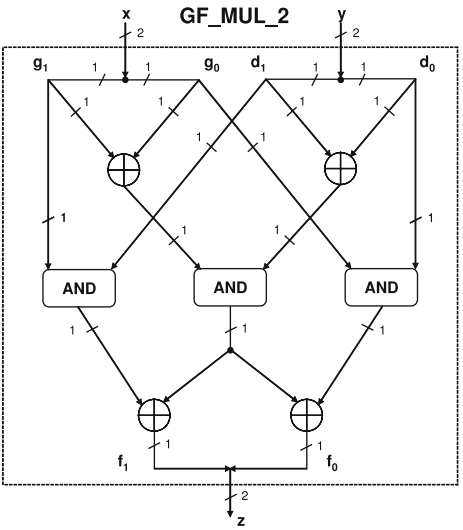

Fig. 10.33 $GF(2^2)$ multiplier, GF_MUL_2.

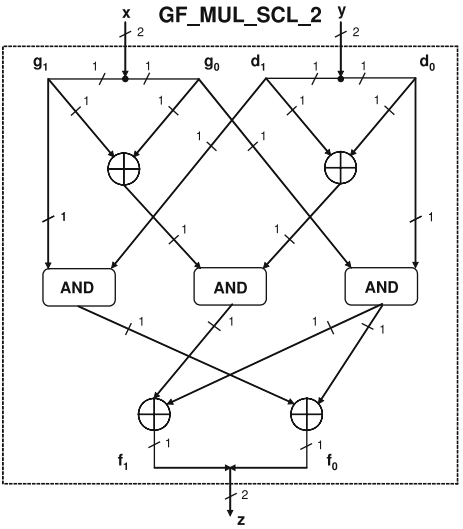

Fig. 10.34 $GF(2^2)$ multiplier and scaler, GF_MUL_SCL_2. It performs multiplication Nxy, where x, y, N $\in GF(2^2)$, x and y are input variables, and N is a constant.

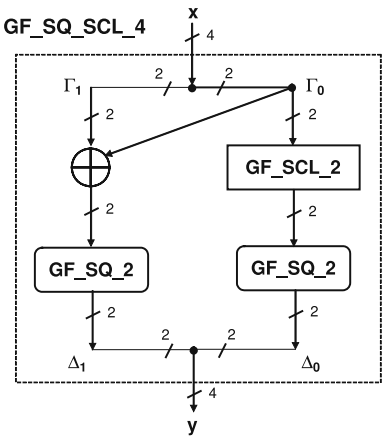

Fig. 10.35 $GF(2^4)$ squarer and scaler, GF_SQ_SCL_4. It performs operation νx^2, where x, $\nu \in GF(2^4)$, x is an input variable, and ν is a constant, $\nu = 0 \cdot z + N^2$. Notation: GF_SQ_2—$GF(2^2)$ squarer (see Figure 10.31) and GF_SCL_2—$GF(2^2)$ scaler (see Figure 10.37).

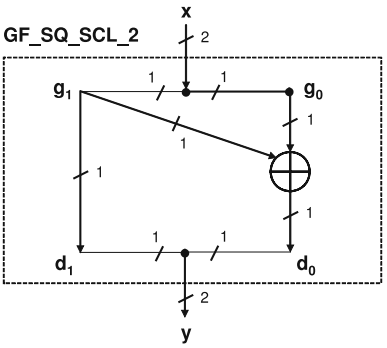

Fig. 10.36 $GF(2^2)$ squarer and scaler, GF_SQ_SCL_2. It performs operation Nx^2, where x, N $\in GF(2^2)$, x is an input variable, and N is a constant.

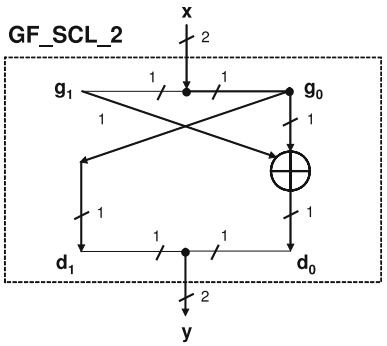

Fig. 10.37 $GF(2^2)$ scaler, GF_SCL_2. It performs operation Nx, where x, N $\in GF(2^2)$, x is an input variable, and N is a constant.

The exact forms of matrices X^{-1}, $(MX)^{-1}$, MX, and X are given by Equations 10.29–10.32.

$$X^{-1} = \begin{bmatrix} 1&1&1&0&0&1&1&1 \\ 0&1&1&1&0&0&0&1 \\ 0&1&1&0&0&0&1&1 \\ 1&1&1&0&0&0&0&1 \\ 1&0&0&1&1&0&1&1 \\ 0&0&0&0&0&0&0&1 \\ 0&1&1&0&0&0&0&1 \\ 0&1&0&0&1&1&1&1 \end{bmatrix} \tag{10.29}$$

$$(MX)^{-1} = \begin{bmatrix} 1\,0\,0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,1\,0\,0\,1\,1 \\ 0\,1\,0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,1 \\ 1\,1\,0\,1\,0\,0\,0\,0 \\ 1\,0\,1\,0\,0\,1\,0\,0 \\ 0\,0\,0\,1\,1\,0\,0\,1 \\ 0\,1\,1\,1\,0\,0\,1\,1 \end{bmatrix} \tag{10.30}$$

$$MX = \begin{bmatrix} 0\,0\,1\,0\,1\,0\,0\,0 \\ 1\,0\,0\,0\,1\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,0\,1 \\ 1\,0\,1\,0\,1\,0\,0\,0 \\ 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,0\,1\,1\,0\,1 \\ 0\,0\,1\,1\,0\,0\,1\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \end{bmatrix} \tag{10.31}$$

$$X = \begin{bmatrix} 0\,0\,0\,1\,0\,0\,1\,0 \\ 1\,1\,1\,0\,1\,0\,1\,1 \\ 1\,1\,1\,0\,1\,1\,0\,1 \\ 0\,1\,0\,0\,0\,0\,1\,0 \\ 0\,1\,1\,1\,1\,1\,1\,0 \\ 1\,0\,1\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,1\,0\,0 \end{bmatrix} \tag{10.32}$$

As shown in Figure 10.29, the $GF(2^8)$ inverter can be decomposed into one $GF(2^4)$ inverter, three $GF(2^4)$ multipliers, two $GF(2^4)$ adders (4-bit XOR gates), and one $GF(2^4)$ squarer and scaler. The $GF(2^4)$ inverter (see Figure 10.30) looks almost the same as $GF(2^8)$ inverter, with component operations in $GF(2^4)$ replaced by operations in $GF(2^2)$. The $GF(2^2)$ does not involve any logic, as it is equivalent to squaring and thus to a circular rotation by one bit position, as shown in Figure 10.31.

The $GF(2^4)$ multiplier can be decomposed into two $GF(2^2)$ multipliers, one $GF(2^2)$ multiplier/scaler, and four $GF(2^2)$ adders (2-bit XOR gates), as shown in Figure 10.32. The $GF(2^2)$ multiplier (see Figure 10.33) has the same structure, with the multipliers and the multiplier/scaler in $GF(2^2)$ replaced by AND gates (multipliers in $GF(2)$).

The $GF(2^4)$ squarer and scaler, shown in Figure 10.35, is an optimized version of the circuit that performs squaring in $GF(2^4)$ followed by multiplication by a specially chosen constant, ν (see [3] for more information about the optimum choice of ν). Similarly, the $GF(2^2)$ multiplier and scaler, shown in Figure 10.34, combines multiplication of two variables in $GF(2^2)$ followed by multiplication by a specifically chosen constant N. The $GF(2^2)$ squarer and scaler, shown in Figure 10.36,

combines squaring with multiplication by N. The multiplication by a constant N can be implemented using just one XOR gate, as shown in Figure 10.37.

Without any further optimizations, the $GF(2^8)$ inverter includes 88 two-input XOR gates and 36 two-input AND gates, and its critical path passes through 14 two-input XOR gates and 4 two-input AND gates. In an FPGA implementation based on 4-input look-up tables (LUTs), $GF(2^4)$ inverter, $GF(2^4)$ squarer and scaler, the $GF(2^2)$ multiplier, and the $GF(2^2)$ multiplier and scaler can all be implemented using look-up tables, so no lower level operations are required. The total number of look-up tables (LUTs) required to implement the $GF(2^8)$ inverter becomes 58, and its critical path delay is equal to the delay of eight logic levels (LUTs).

As explained in [3], multiple further optimizations based on resource sharing and optimum gate type choice can be used to further reduce circuit area. The derivation of the minimum-area circuit, together with a detailed justification of design choices can be found in [3, 4]. The appendices of [3] contain the corresponding C program, which can be used as a source of test vectors, and the manually optimized Verilog code, which can be used as a starting point for a hardware implementation.

10.6.2 MixColumns and InvMixColumns

10.6.2.1 Basic Implementation

The *MixColumns* transformation can be expressed as a matrix multiplication in the Galois field $GF(2^8)$ as shown in Equation (10.14). Each symbol in this equation (such as b_i, a_i, 03) represents an 8-bit element of the Galois field. Each byte of the result of a matrix multiplication (10.14) is an XOR of four bytes representing the Galois field product of a byte a_0, a_1, a_2, or a_3 by a respective constant. As a result, the entire *MixColumns* transformation can be performed using two layers of XOR gates, with up to 3-input gates in the first layer and 4-input gates in the second layer. In FPGAs, each of these XOR operations requires only one 4-bit look-up table.

The *InvMixColumns* transformation can be expressed as a matrix multiplication in the Galois field $GF(2^8)$ as shown in Equation (10.15).

The primary differences, compared to *MixColumns*, are the larger hexadecimal values of the matrix coefficients. Multiplication by these constant elements of the Galois field leads to the more complex dependence between the bits of a variable input and the bits of a respective product.

The entire *InvMixColumns* transformation can be performed using two layers of XOR gates, with up to 6-input gates in the first layer and 4-input gates in the second layer. Because of the use of gates with larger number of inputs, the *InvMixColumns* transformation has a longer critical path compared to the *MixColumns* transformation, and as a result, the decryption circuit imposes a limit on the minimum clock period of the entire encryption/decryption unit.

10.6.2.2 Implementations with Resource Sharing

Coefficients of $d(X)$ are more complex than coefficients of $c(X)$; therefore, decryption is always slower than encryption [8]. Moreover, hardware structures implementing *InvMixColumns* are always larger. To reduce hardware cost, the *InvMixColumns* matrix can be decomposed in such a way that some portion of the hardware will be re-used for *MixColumns* implementation. Since *MixColumns* and *InvMixColumns* functions are defined on 32-bit words, we will call this decomposition the word-level resource sharing. There are two possible approaches: parallel and serial *InvMixColumns* decomposition. In addition to word-level sharing, resources can be shared on a byte level and on a bit level [14].

Parallel InvMixColumns Decomposition

Parallel *InvMixColumns* decomposition was first proposed by J. Wolkerstorfer in [43]. It is based on the observation that $d(X)$ can be expressed using $c(X)$ in the following way:

$$d(X) = c(X) + e(X) \tag{10.33}$$

where $e(X)$ is an extension polynomial defined as

$$e(X) = x^3X^3 + (x^3 + x^2)X^2 + x^3X + (x^3 + x^2) \tag{10.34}$$

or in hexadecimal format

$$e(X) = \{08\}X^3 + \{0C\}X^2 + \{08\}X + \{0C\} \tag{10.35}$$

Equation (10.15) can thus be written in the form

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = ([C] + [E]) \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.36}$$

where

$$[C] = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \tag{10.37}$$

and

$$[E] = \begin{bmatrix} 0C\ 08\ 0C\ 08 \\ 08\ 0C\ 08\ 0C \\ 0C\ 08\ 0C\ 08 \\ 08\ 0C\ 08\ 0C \end{bmatrix} \tag{10.38}$$

When both *MixColumns* and *InvMixColumns* have to be implemented in the same piece of hardware, the matrix $[E]$ from Equation (10.36) is added to the matrix $[C]$ only during decryption, and thus the matrix $[C]$ is shared by both encryption and decryption processes. Implementation of such a structure in the hardware can be viewed as implementation of four identical blocks A (see Figure 10.38), each giving an output of one byte of $b(X)$. Inputs of these blocks are permuted in the same way as coefficients of the *MixColumns* (or *InvMixColumns*) matrix (see Figure 10.38b).

Serial InvMixColumns Decomposition

MixColumns and *InvMixColumns* are derived as mutual inverses, and therefore, they are related such that $c(X) \cdot d(X) = 1$. There exists another relationship between $c(X)$ and $d(X)$: the inverse $d(X)$ of the polynomial $c(X)$ in the ring R is given by the formula

$$d(X) = c^{-1}(X) = c^3(X) \tag{10.39}$$

Equation (10.39) suggests that the *InvMixColumns* operation can be realized by repeating *MixColumns* three times. For hardware implementations, Equation (10.39) can be expressed as

$$d(X) = c(X) \cdot c^2(X) \tag{10.40}$$

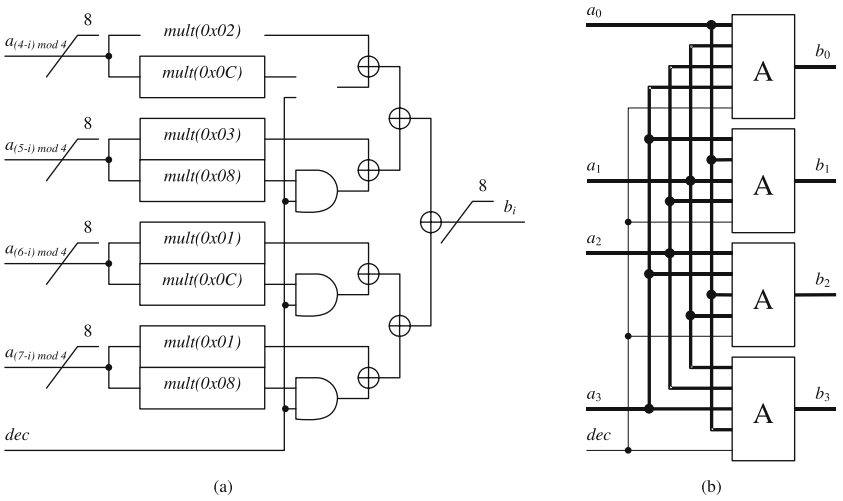

Fig. 10.38 Resource sharing based on parallel *InvMixColumns* decomposition: (**a**) line multiplication block A, (**b**) overall structure.

where

$$c^2(X) = x^2X^2 + x^2 + 1 \tag{10.41}$$

Therefore, the *InvMixColumns* function can be implemented using the *MixColumns* function and the $c^2(X)$ polynomial. The $c^2(X)$ polynomial can be expressed with coefficients in hexadecimal format:

$$c^2(X) = \{00\}X^3 + \{04\}X^2 + \{00\}X + \{05\} \tag{10.42}$$

Following Equation (10.40) the Equation (10.15) can be expressed as

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = ([C] \cdot [F]) \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.43}$$

where $[C]$ has the same meaning as in Equation (10.37) and

$$[F] = \begin{bmatrix} 05\ 00\ 04\ 00 \\ 00\ 05\ 00\ 04 \\ 04\ 00\ 05\ 00 \\ 00\ 04\ 00\ 05 \end{bmatrix} \tag{10.44}$$

Comparing $c(X)$ and $c^2(X)$ we see that $c^2(X)$ is much simpler in implementation than $c(X)$ because two of its coefficients are equal to zero. Multiplication of matrices represents a serial arrangement of corresponding modules with common input (see Figure 10.39a) or common output (see Figure 10.39b). Using the same approach as for the parallel *InvMixColumns* decomposition, the hardware structure with a common input can be implemented using four instances of two types of blocks A and B (see Figure 10.40). This structure implements multiplication of the 4-byte input by one line of $c(X)$ and $c^2(X)$.

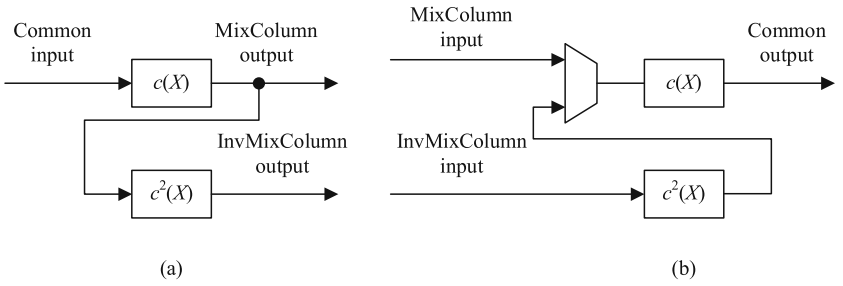

Fig. 10.39 Resource sharing based on serial *InvMixColumns* decomposition with (**a**) common input, (**b**) common output.

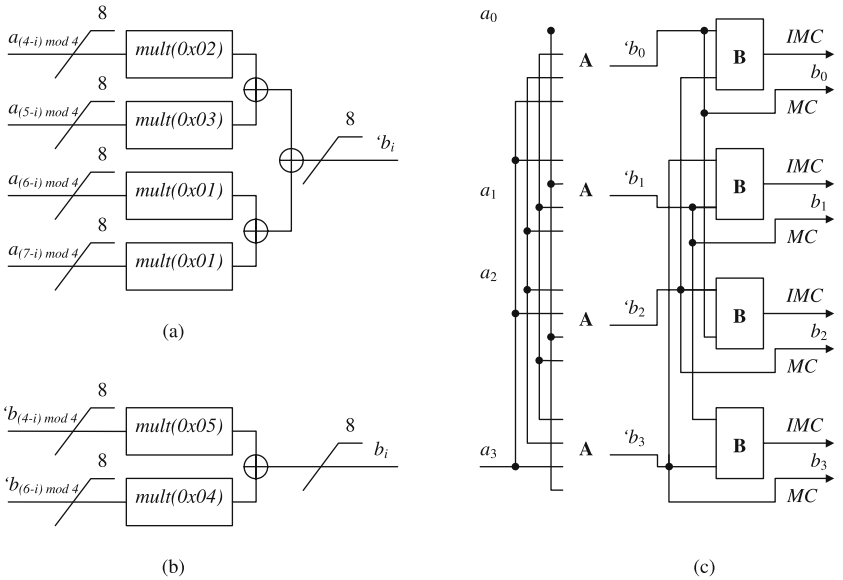

Fig. 10.40 Serial *InvMixColumns* decomposition with common input: (**a**) line multiplication block A, (**b**) line multiplication block B, (**c**) overall structure.

10.7 Hardware Architectures of a Single Round of AES

10.7.1 S-Box-Based Architecture

The block diagrams of the encryption/decryption unit based on S-boxes in the basic iterative architecture is shown in Figure 10.41. Only register R1 is present in the basic iterative architecture. The best placement for this register is either before or after the combined *SubBytes* and *InvSubBytes* transformation, where encryption and decryption data paths converge. The critical path is located in the decryption circuit and includes *InvShiftRows* (interconnects), *AddRoundKey* (XOR operation), *InvMixColumns*, a 3-to-1 multiplexer, and *InvSubBytes*. In this architecture, 11, 13, and 15 clock cycles are required in order to process one block of data for 128-, 192-, and 256-bit keys, respectively.

In Figure 10.42 several registers have been added in order to create an architecture with two stages of inner-round pipelining. The location of these registers has been chosen in such a way to divide the critical path into two approximately equal paths. This way the minimum clock period should be equal to approximately half of the clock period for the basic iterative architecture, allowing processing of data with approximately twice as high throughput in non-feedback cipher modes.

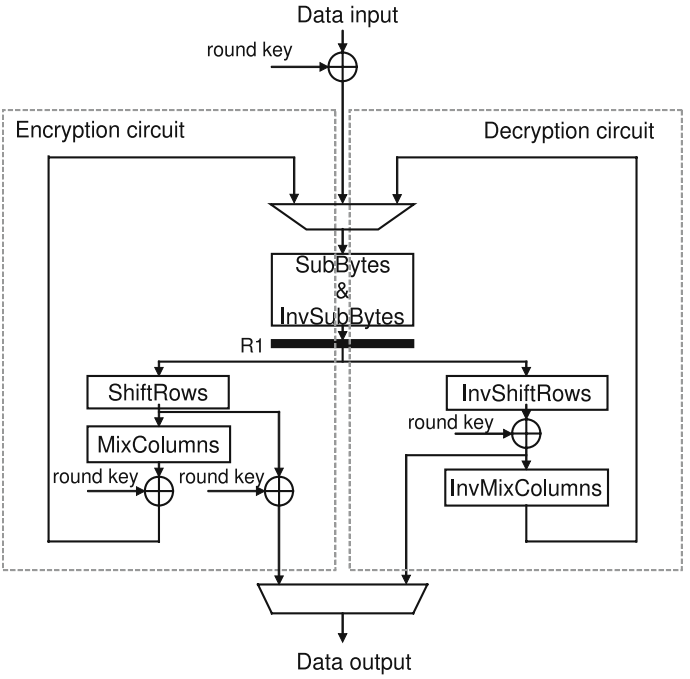

Fig. 10.41 Typical S-box-based AES basic iterative architecture.

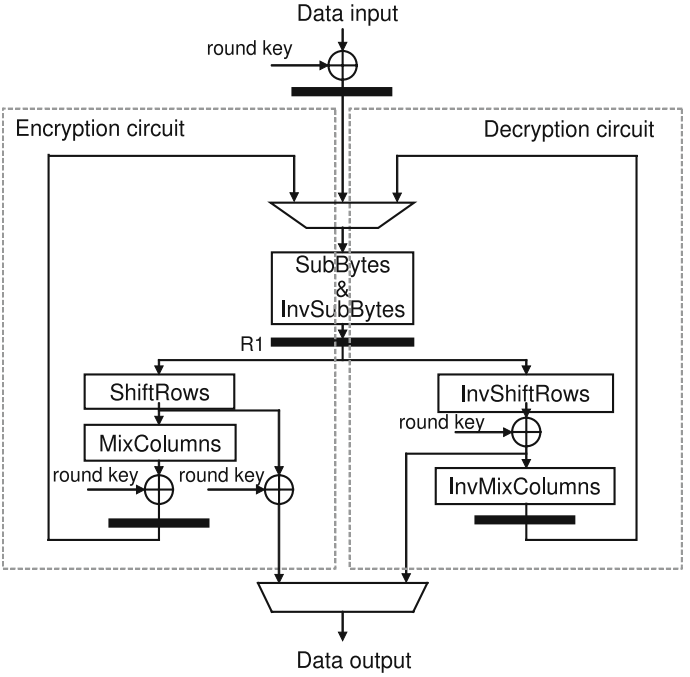

Fig. 10.42 S-box-based AES architecture with two stages of inner-round pipelining.

10.7.2 T-Box-Based Architecture

T-box-based algorithm for implementing AES was first proposed in [7] for a software implementation using 32-bit microprocessors. In [13], this approach was adapted for hardware implementations.

The T-box approach allows the computation of the entire round of AES using only table look-ups and XOR operations. In Figure 10.43, we show the mathematical description of AES round operations. This representation leads to the derivation of the T-box representation of an AES encryption round, as shown in Figure 10.44. The precomputed tables, called T-boxes, represent the combined application of the *SubBytes* and *MixColumns* transformations. They are defined as follows:

$$T_0[a] = \begin{bmatrix} 02 \cdot S[a] \\ S[a] \\ S[a] \\ 03 \cdot S[a] \end{bmatrix} \qquad T_1[a] = \begin{bmatrix} 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \\ S[a] \end{bmatrix} \tag{10.45}$$

$$T_2[a] = \begin{bmatrix} S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \end{bmatrix} \qquad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \end{bmatrix} \tag{10.46}$$

SubBytes

$$\mathbf{b}_{i,j} = \mathbf{S}[\mathbf{a}_{i,j}]$$

ShiftRows

$$\begin{bmatrix} \mathbf{c}_{0,j} \\ \mathbf{c}_{1,j} \\ \mathbf{c}_{2,j} \\ \mathbf{c}_{3,j} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_{0,j} \\ \mathbf{b}_{1,j+1} \\ \mathbf{b}_{2,j+2} \\ \mathbf{b}_{3,j+3} \end{bmatrix}$$

sums mod 4

MixColumns

$$\begin{bmatrix} \mathbf{d}_{0,j} \\ \mathbf{d}_{1,j} \\ \mathbf{d}_{2,j} \\ \mathbf{d}_{3,j} \end{bmatrix} = \begin{bmatrix} \mathbf{02} & \mathbf{03} & \mathbf{01} & \mathbf{01} \\ \mathbf{01} & \mathbf{02} & \mathbf{03} & \mathbf{01} \\ \mathbf{01} & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ \mathbf{03} & \mathbf{01} & \mathbf{01} & \mathbf{02} \end{bmatrix} \begin{bmatrix} \mathbf{c}_{0,j} \\ \mathbf{c}_{1,j} \\ \mathbf{c}_{2,j} \\ \mathbf{c}_{3,j} \end{bmatrix}$$

AddRoundKey

$$\begin{bmatrix} \mathbf{e}_{0,j} \\ \mathbf{e}_{1,j} \\ \mathbf{e}_{2,j} \\ \mathbf{e}_{3,j} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_{0,j} \\ \mathbf{d}_{1,j} \\ \mathbf{d}_{2,j} \\ \mathbf{d}_{3,j} \end{bmatrix} \oplus \begin{bmatrix} \mathbf{k}_{0,j} \\ \mathbf{k}_{1,j} \\ \mathbf{k}_{2,j} \\ \mathbf{k}_{3,j} \end{bmatrix}$$

Fig. 10.43 Mathematical description of AES round operations.

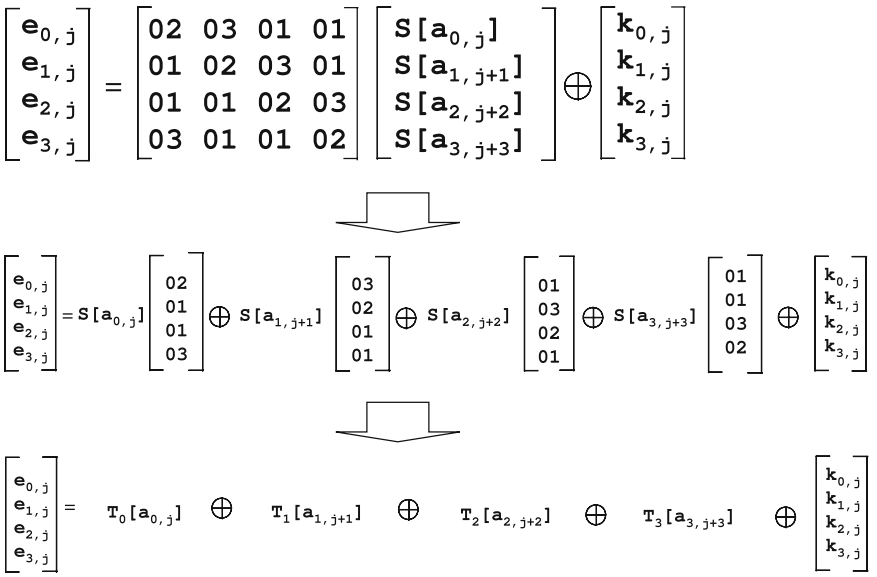

Fig. 10.44 Mathematical derivation of the T-box representation of an AES encryption round.

Compared to the S-box tables, which are of the size of 8×8 bits, the T-box tables are of the size of 8×32 bits. The entire 32-bit column of an output of a single round of AES, e_j, can be computed using the following formula:

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}] \oplus K_j \tag{10.47}$$

where T_0, T_1, T_2, T_3 are the precomputed 8×32-bit look-up tables, and K_j is a jth word of a round key K. All indices $j+1$, $j+2$, $j+3$ are computed modulo 4. For example,

$$e_2 = T_0[a_{0,2}] \oplus T_1[a_{1,3}] \oplus T_2[a_{2,0}] \oplus T_3[a_{3,1}] \oplus K_2 \tag{10.48}$$

In Figure 10.45, we show in a graphical form an example of computing the value of the column e_2 of the encryption round output using T-box approach.

Since *MixColumns* operation is not performed in the last round of encryption, the last round needs to be treated in a special way. In this round, S-boxes need to be used instead of T-boxes. Fortunately, no additional memory space is needed to implement S-boxes, as 1-byte outputs of the S-box transformation can be easily extracted from the 4-byte outputs of the T-box transformation corresponding to the same 1-byte input. For example,

$$S[a] = byte(1, T_0[a]) = byte(2, T_1[a]) = byte(3, T_2[a]) = byte(0, T_3[a]) \tag{10.49}$$

where $byte(n, X)$ represents the nth byte of a variable X.

In Figure 10.46, a block diagram of the encryption function based on the use of T-box look-up tables is shown. The encryption input consists of 16 bytes, arranged as follows:

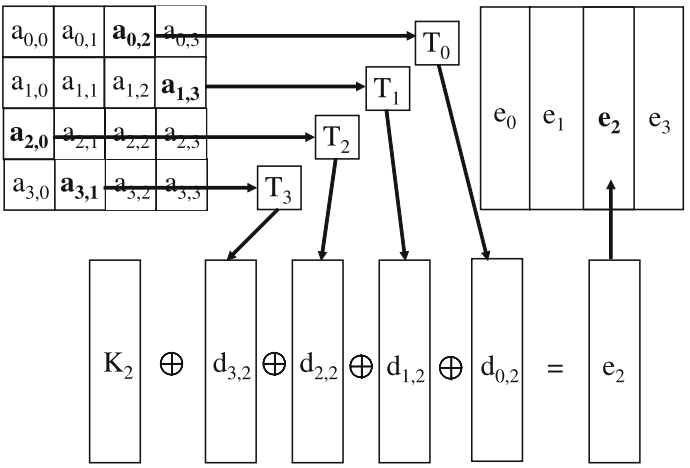

Fig. 10.45 An example of computing the value of the column e_2 of the encryption round output using T-box approach.

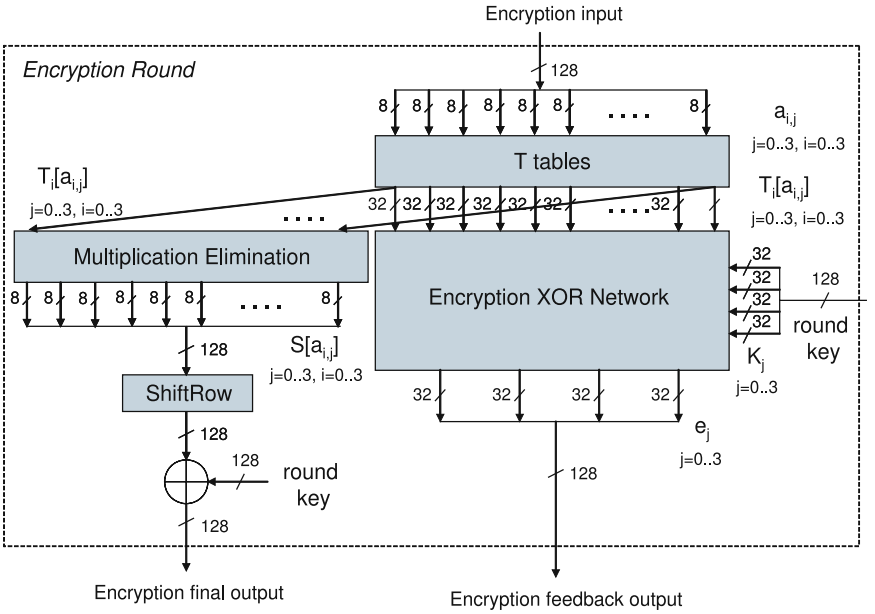

Fig. 10.46 Block diagram of the T-box-based AES encryption round.

$$a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, a_{1,1}, a_{2,1}, a_{3,1}, a_{0,2}, a_{1,2}, a_{2,2}, a_{3,2}, a_{0,3}, a_{1,3}, a_{2,3}, a_{3,3} \tag{10.50}$$

Each of these bytes is an input to the appropriate look-up table $T_i[a_{i,j}]$, with an index the same as a row index of the byte $a_{i,j}$. Sixteen such look-up tables working in parallel form a functional block called *T tables*. *Encryption XOR Network* is a block

that based on 16 T-box tables outputs $T_i[a_{i,j}]$ ($j = 0...3$, $i = 0...3$) and four words of the round key K, K_j ($j = 0...3$) computes four 32-bit columns of the encryption round output e_j. The exact dependence between inputs and outputs of this block is given by Equation 10.47. Finally, the *Multiplication Elimination* extracts values of the S-box outputs $S[a_{i,j}]$ based on the values of the T-box outputs $T_i[a_{i,j}]$. The operation of this block is given by Equation 10.49, and the block itself does not involve any logic, just routing.

A similar derivation can be performed in order to represent decryption using a separate set of inverse T-boxes, ${T_0}^{-1}$, ${T_1}^{-1}$, ${T_2}^{-1}$, ${T_3}^{-1}$. These inverse T-box tables are defined as follows:

$$T_0^{-1}[a] = \begin{bmatrix} 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \end{bmatrix} \qquad T_1^{-1}[a] = \begin{bmatrix} 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \end{bmatrix} \tag{10.51}$$

$$T_2^{-1}[a] = \begin{bmatrix} 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \end{bmatrix} \qquad T_3^{-1}[a] = \begin{bmatrix} 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \end{bmatrix} \tag{10.52}$$

The equation describing an output of the decryption round is given below:

$$d_j = T_0^{-1}[a_{0,j}] \oplus T_1^{-1}[a_{1,j+3}] \oplus T_2^{-1}[a_{2,j+2}] \oplus T_3^{-1}[a_{3,j+1}] \oplus IMC(K_j) \tag{10.53}$$

where T_0^{-1}, T_1^{-1}, T_2^{-1}, T_3^{-1} are the precomputed 8×32-bit look-up tables, K_j is a jth word of a round key K, and *IMC* is the *InvMixColumns* transformation. All indices $j+3$, $j+2$, $j+1$ are computed modulo 4.

In the last round of decryption the *InvMixColumns* operation is not executed. As a result, the outputs of the *InvSubBytes* transformation, $S^{-1}[a]$, must be computed. In this case, however, the computation of $S^{-1}[a]$ as a function of $T_i^{-1}[a]$ requires some extra logic that implements multiplication by a constant in $GF(2^8)$ [13]. For example, given the value of $T_0^{-1}[a]$, $S^{-1}[a]$ can be computed as follows:

$$\begin{aligned} S^{-1}[a] &= 0E^{-1} \cdot byte(0, T_0^{-1}[a]) = 09^{-1} \cdot byte(1, T_0^{-1}[a]) \\ &= 0D^{-1} \cdot byte(2, T_0^{-1}[a]) = 0B^{-1} \cdot byte(3, T_0^{-1}[a]) \\ &= E5 \cdot byte(0, T_0[a]) = 4F \cdot byte(1, T_0[a]) \\ &= E1 \cdot byte(2, T_0[a]) = C0 \cdot byte(3, T_0[a]) \end{aligned} \tag{10.54}$$

where $byte(n, X)$ represents the nth byte of a variable X. Based on the above equations, each bit of $S^{-1}[a]$ can be computed using four different equations, each giving exactly the same value. As a result, for each bit, we can choose an equation with the smallest number of terms. If we do that, we can express the bits of $S^{-1}[a] = (s_7^{-1} s_6^{-1} s_5^{-1} s_4^{-1} s_3^{-1} s_2^{-1} s_1^{-1} s_0^{-1})$ as follows:

$$
\begin{aligned}
x &= byte(0, T_0^{-1}[a]) \\
y &= byte(1, T_0^{-1}[a]) \\
z &= byte(3, T_0^{-1}[a])
\end{aligned} \tag{10.55}
$$

$$
\begin{aligned}
s_7^{-1} &= y_7 \oplus y_4 \oplus y_1 \\
s_6^{-1} &= y_6 \oplus y_3 \oplus y_0 \\
s_5^{-1} &= y_5 \oplus y_2 \\
s_4^{-1} &= y_4 \oplus y_1 \\
s_3^{-1} &= z_3 \oplus z_2 \oplus z_1 \\
s_2^{-1} &= x_6 \oplus x_5 \oplus x_0 \\
s_1^{-1} &= x_7 \oplus x_5 \oplus x_4 \\
s_0^{-1} &= y_5 \oplus y_2 \oplus y_0
\end{aligned} \tag{10.56}
$$

For the computations using outputs from tables T_1^{-1}, T_2^{-1}, and T_3^{-1}, the input word must be rotated by one, two, and three byte positions, respectively, before applying the same transformation. This rotation is equivalent to defining variables x, y, and z as follows:

$$
\begin{aligned}
x &= byte(1, T_1^{-1}[a]) = byte(2, T_2^{-1}[a]) = byte(3, T_3^{-1}[a]) \\
y &= byte(2, T_1^{-1}[a]) = byte(3, T_2^{-1}[a]) = byte(0, T_3^{-1}[a]) \\
z &= byte(0, T_1^{-1}[a]) = byte(1, T_2^{-1}[a]) = byte(2, T_3^{-1}[a])
\end{aligned} \tag{10.57}
$$

and then applying transformation given by Equation 10.56. The entire *Decryption Round Circuit* is shown in Figure 10.47. The functional block T^{-1} *tables* consists of sixteen 8×32-bit look-up tables working in parallel. The operation of the *Decryption XOR Network* is given by Equation 10.53 and the operation of the *Inverse Multiplication Elimination* is given by Equation 10.55, 10.57, and 10.56.

In Figure 10.48, a circuit capable of performing both encryption and decryption using T-box approach is shown. The exact location of the register may depend on specific technology. For example, in Xilinx FPGAs, T tables are likely to be implemented using Block RAMs, which have synchronous outputs. Thus, the register would need to be placed at the output of the T *tables* and T^{-1} *tables* blocks, inside of the encryption and decryption rounds, shown in Figures 10.46 and 10.47, respectively.

Independently of the exact location of the register, the critical path is likely to be of the same length for the encryption and decryption circuits and includes two functional blocks: T/T^{-1} *tables* and *Encryption/Decryption XOR Network*.

Compared to the S-box-based implementation, the T-box-based implementation requires four times larger memory space, but fewer logic resources. In the T-box implementation the critical path is longer for encryption, but shorter for decryption,

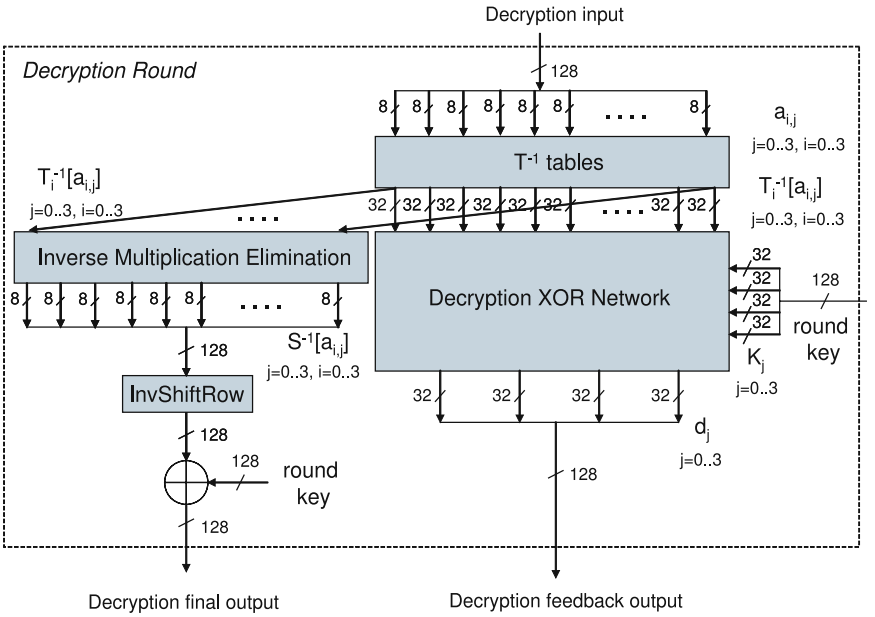

Fig. 10.47 Block diagram of the T-box-based AES decryption round.

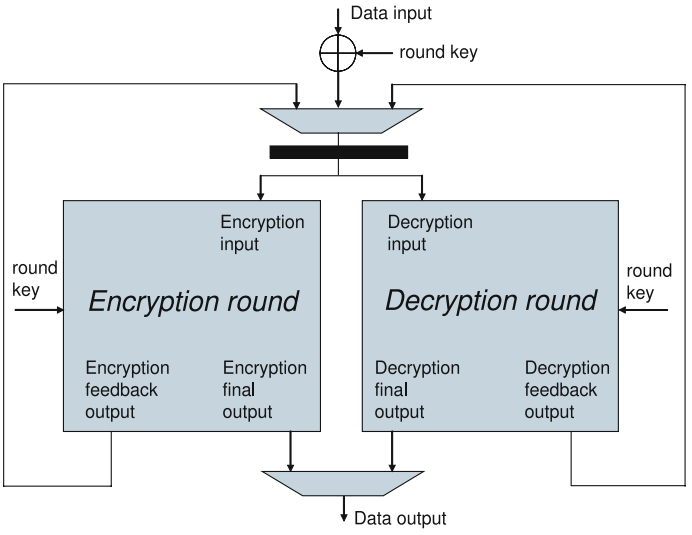

Fig. 10.48 Block diagram of the T-box-based AES encryption/decryption circuit.

compared to the S-box-based implementation. For example, in [13], the T-box-based approach was shown to produce a 8% speed-up for decryption, a 22% slowdown for encryption, and a 22% speed-up for encryption/decryption (a single-clock circuit capable of performing both operations) vs. equivalent implementations following the

S-box approach. All compared implementations targeted Altera FPGAs. The exact ratios of the costs and speeds for both approaches depend on the choice of a specific technology (FPGA vs. ASIC, specific FPGA family, specific ASIC standard-cell library, etc.) used for the implementation.

10.7.3 Compact Architectures

The AES round shown in Figure 10.49 reveals a great deal of parallelism. The data bytes are ordered from the most significant (byte 0) to the least significant (byte F) assuming big-endian representation. The round is composed of sixteen 8-bit S-boxes computing *SubBytes* and four 32-bit *MixColumns* operations, working independent of each other. The only operation that spans throughout the entire 128-bit block is *ShiftRows*.

It is possible to implement only four *SubBytes* and one *MixColumns* in order to compact the AES implementation. Ideally, the resources should be cut by four, while execution of one round should take four clock cycles. This approach would result in approximately four times lower performance than for the basic architecture.

Cutting the resources by 75% may not appear easy. The *folded round*, as we call the modified round, still must transform 128 bits, and storage for all 128 bits of the data block must exist. Another complication is related to the implementation of the *ShiftRows* operation. The data bytes processed in the AES round cannot return to the same positions in the block register because it would not execute the *ShiftRows* operation. On the other hand, those same bytes cannot be placed into locations indicated by *ShiftRows* because those locations are occupied by other bytes that have not yet been processed. Therefore, additional bits of intermediate results must be stored, and more logic resources are needed.

One of the possible architectures for a folded implementation is shown in Figure 10.50a. This architecture requires one 128-bit register, one 96-bit register, and one 32-bit-wide 4-to-1 multiplexer on top of the main cipher operations. The multiplexer becomes even bigger when both *ShiftRows* and *InvShiftRows* are

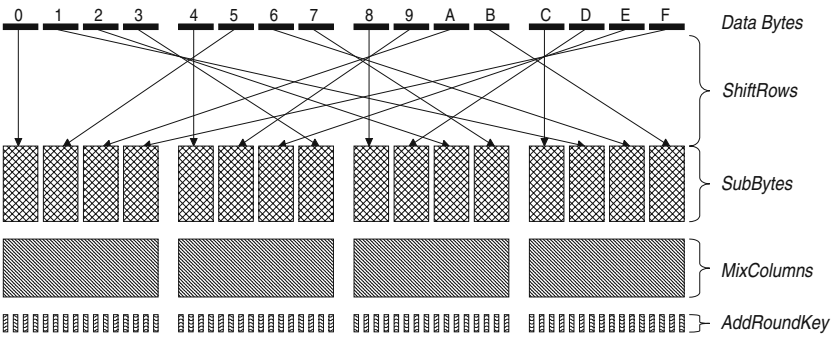

Fig. 10.49 Operations within AES encryption round.

implemented using same logic resources. The execution of one round takes four clock cycles. The authors believe that this, or very similar architecture, was implemented by A. Satoh et al. [33]. Their results show that the 4-cycle round takes 50% of the resources required by the 1-cycle round and yields four times lower throughput.

Another possible architecture is shown in Figure 10.50b. The 96-bit register is implemented as three 32-bit registers inserted into round operations creating a pipeline. In the case of FPGAs, those 32-bit registers will most likely be placed in the same Slices as logic operations yielding better resource utilization. The critical path is also shortened which permits the execution at a higher clock rate; however, the execution of the entire round requires seven, instead of four, clock cycles. The authors believe that this architecture was implemented by S. McMillan et al. [27], but no sufficient details are provided in this chapter. S. McMillan et al. reported only slight difference of 48 Slices (16%), and large difference of 24 Block RAMs (75%), between one-round unrolled and folded architecture.

10.7.3.1 Folded Register

The two folded architectures described above are very straightforward and resulted in small logic savings. In order to create a folded architecture with better parameters, we need to explore fine details of FPGA devices. Let us arrange data bytes into rows as shown in Figure 10.51. This data arrangement is consistent with a *state* introduced in [8]. The following exercise can now be executed in steps:

1. Read input bytes: 0, 5, A, F; execute *SubBytes*, *MixColumns*, and *AddRoundKey* on them; write results to the output at locations 0, 1, 2, 3. This step is highlighted in Figure 10.51.
2. Repeat above operations for input bytes 4, 9, E, 3; write results at output locations 4, 5, 6, 7.
3. Repeat above operations for bytes 8, D, 2, 7; write results at locations 8, 9, A, B.
4. Repeat above operations for bytes C, 1, 6, B; write results at locations C, D, E, F. Output now becomes input for the next step.

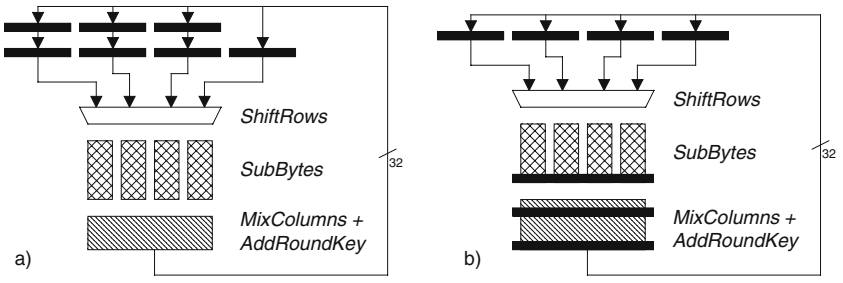

Fig. 10.50 Folded architectures (**a**) by A. Satoh et al. [33]; (**b**) by S. McMillan et al. [27].

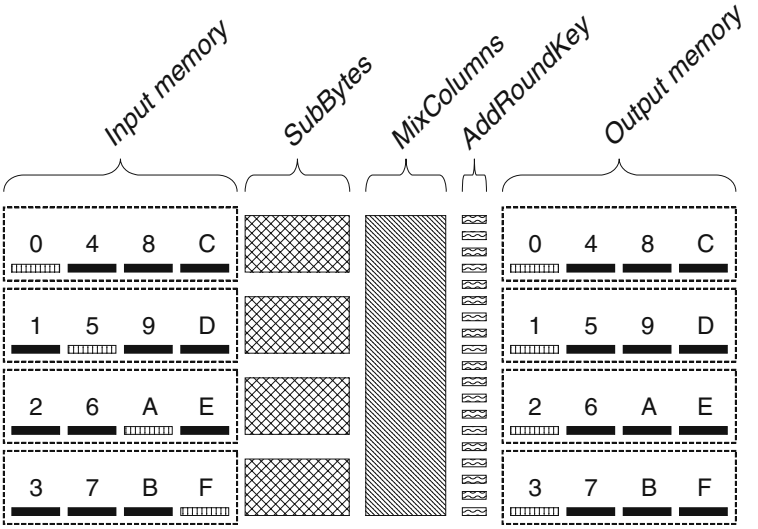

Fig. 10.51 Data arrangement in the folded architecture. Data bytes involved in the first step of calculation are highlighted.

In those four steps the entire AES round was executed including *ShiftRows* operation. At each step only one byte was read from each input row, and one byte was written to each output row. A similar exercise with identical conclusions can be executed for decryption transformation. Each row can be viewed as an addressable 8-bit-wide memory. The correct execution of *ShiftRows* and *InvShiftRows* is now resolved to the proper addressing of each of the memories at the consecutive clock cycles. At the fourth clock cycle output memories become input memories and vice versa.

10.7.3.2 FPGA Dual-Port RAM-Based Implementation

Each CLB Slice in Xilinx FPGAs contains two look-up tables (LUT), which are the primary resources for logic implementation. Typically LUTs are configured as small 16×1 ROM tables implementing logic functions of up to four inputs; however, other configurations are also possible. Two LUTs within the same Slice can implement a 16×1 dual-port RAM. An 8-bit-wide dual-port RAM can be implemented using eight CLB Slices. This memory can be divided into two banks, each addressed by a different port. One port is used for reading data from the memory, while the other one writes results back to the same memory. The switching between banks can be achieved by flipping one address bit in both ports every fourth clock cycle.

The dual-port RAM-based solution has major advantages over solutions presented in Figure 10.50:

- The logic resources required for storing intermediate results are far smaller.
- The multiplexer used before for *ShiftRows* and *InvShiftRows* is no longer needed.

- The complicated routing resulting from implementation of *ShiftRows* and *InvShiftRows* is avoided, yielding better performance.

10.7.3.3 FPGA Shift Register-Based Implementation

A better solution may result from the following observation: all bytes from the output of *AddRoundKey* are written into consecutive locations in the output memory in consecutive clock cycles. Therefore, we could use a simple shift register to shift computed data in without generating any addresses. Fortunately, LUTs can also be configured as 16-bit shift registers with variable taps, as shown in Figure 10.52. Four Slices can implement an 8-bit-wide, 16-bit-long shift register. The input of the shift register is used for shifting results in while the output, selected dynamically by changing tap address, is used for reading data out. This solution encompasses all of the advantages of the dual-port RAM-based solution and requires less than half of the logic resources than the dual-port RAM.

10.7.3.4 Encryption/Decryption Unit

The optimized circuit is capable of performing encryption and decryption. The AES encryption and decryption rounds substantially differ from the point of view of hardware implementations. One of the inconveniences arises from the fact that the *AddRoundKey* is executed after *MixColumns* in the case of encryption and before *InvMixColumns* in the case of decryption. Therefore, a switching logic is required to select appropriate data paths, which affect the performance, as shown in Figure 10.53.

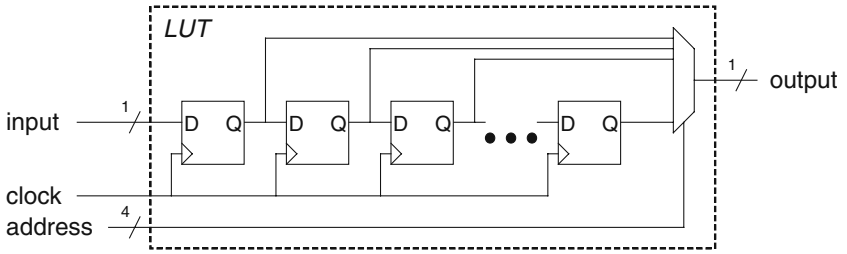

Fig. 10.52 Look-up table (LUT) configured as a shift register.

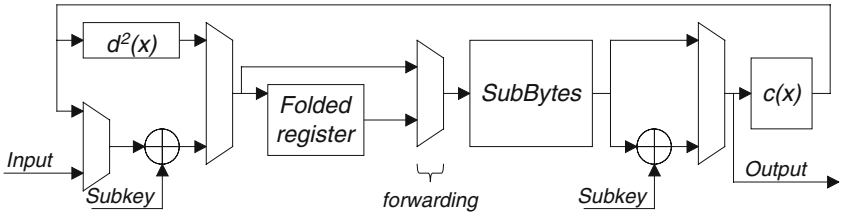

Fig. 10.53 Implementation of the encryption/decryption unit.

10.8 Implementation of Key Scheduling

An AES key scheduling unit can either generate round keys on the fly or it can store them in an internal key memory during the key setup phase and then read them from this memory whenever they are required by the encryption/decryption unit.

An AES key scheduling unit can support just one external key size, e.g., 128-bit key, or it can support all three key sizes described in the standard, i.e., 128-, 192-, and 256-bit keys. In the latter case, the unit is referred to as a 3-in-1 design.

Both kinds of units can be constructed in such a way that they produce 32 bits (one word = $\frac{1}{4}$th of a round key), 64 bits (two words = $\frac{1}{2}$ of a round key), or 128 bits (the entire round key) per clock cycle.

In the basic iterative architecture, only the last of these three designs allows the generation of round keys on the fly. The remaining designs require the key setup phase, during which the round keys are computed and stored in internal memory.

As an example, in Figure 10.54, we present a 3-in-1 key scheduling unit that produces 64-bits (a half of a round key) per clock cycle. The operation of the circuit is described by formulas given in Figure 10.54b that follow the pseudocode from Figure 10.16. The unit is capable of computing two 32-bit words of the key material (k_i and k_{i+1}) per one clock cycle, independently of the size of the main key.

Since each round key is 128-bit long (the size of the input block), two clock cycles are required to calculate each round key. Therefore, this key scheduling unit is not designed for computing subkeys on the fly. Instead, all round keys corresponding to the new main key are computed in advance and stored in the key memory. This computation can be performed in parallel with encrypting data using previous main key, therefore key scheduling does not impose any performance penalty.

The implementation of the key schedule suitable for a more compact encryption/decryption architecture supporting only 128-bit AES keys is shown in Figure 10.55. This architecture computes 32 bits of the key material per clock cycle, therefore, full key schedule execution for a 128-bit key takes 44 clock cycles. The computed round keys are stored in RAM.

The key schedule uses *SubBytes* operation that is identical to the one used in the encryption circuit. Since key schedule does not have to work simultaneously with the encryption unit, it is possible to time share S-boxes between both circuits.

10.9 Optimum Choice of a Hardware Architecture for AES

The choice of an optimum hardware architecture for AES depends on the following major factors:

1. Optimization criteria, such as minimum area, minimum power consumption, maximum throughput, maximum throughput to area ratio, etc.

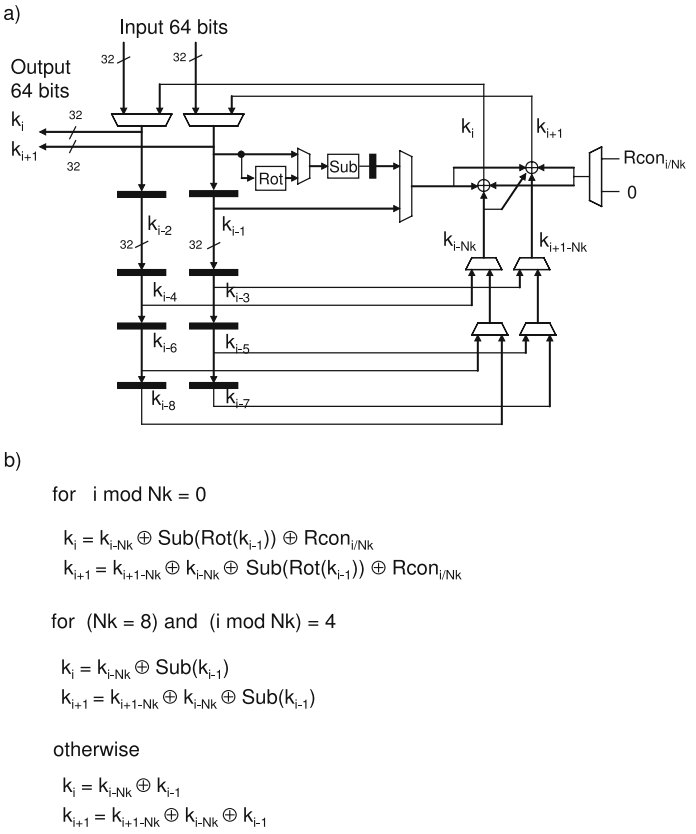

Fig. 10.54 The 3-in-1 key scheduling unit of AES: (**a**) main circuit, (**b**) formulas describing the operation of the circuit.

2. Support for feedback modes of operation, such as CBC and CFB, or non-feedback modes of operation, such as ECB and CTR modes.
3. Support for AES encryption only (e.g., in the block cipher modes of operation that require encryption only, such as CTR and CFB modes) or encryption and decryption (e.g., in the modes that require both operations, such as ECB and CBC).
4. Semiconductor technology of choice, such as ASIC or FPGA.
5. Resistance to side channel attacks, such as differential power analysis, timing analysis, etc.

In case area and/or power consumption are primary concerns, compact architectures described in Section 10.7.3 or the basic iterative architecture, described in Section 10.5.2, should be considered. This choice is independent of the requirements for feedback vs. non-feedback cipher modes and encryption only vs. encryption/decryption. S-box-based architectures will be preferred in this case, and

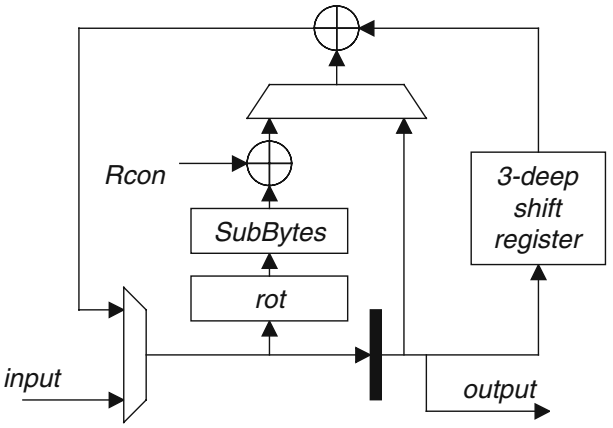

Fig. 10.55 Implementation of the key schedule.

S-boxes may be implemented using logic only (especially in ASIC implementations). In case both encryption and decryption are required, the resource sharing between *MixColumns* and *InvMixColumns*, based on the parallel or serial *InvMixColumns*, decomposition should be considered, as described in Section 10.6.2.

In case the maximum throughput is of primary concern, the choice of the hardware architecture depends on the operating modes that need to be supported.

As shown in Figure 10.20, the basic iterative architecture of AES assures the maximum throughput to area ratio for feedback operating modes such as CBC and CFB. It also guarantees near-optimum throughput and near-optimum area for these operating modes. Therefore, it is very likely to be commonly used in many practical implementations of AES targeting feedback cipher modes. In case only encryption needs to be supported (in the operating modes such as CFB), S-box-based architecture, described in Section 10.7.1, is preferred. In case both encryption and decryption need to be supported (e.g., in the CBC mode) the T-box-based architecture, described in Section 10.7.2, assures the maximum overall clock speed and data throughput.

In the non-feedback cipher modes of operation, such as counter mode, the architecture with the mixed inner- and outer-round pipelining, described in Section 10.5.4, offers the maximum circuit throughput. The S-box-based architecture with S-boxes implemented using logic only (see Sections 10.7.1 and 10.6.1) leads to the highest clock frequency. The throughput in the full mixed inner- and outer-round pipelining is given by

$$Throughput_{full\_mixed} = \frac{block\_size}{T_{CLK_{inner\_round}}(k_{opt})} \tag{10.58}$$

where $T_{CLK_{inner\_round}}(k_{opt})$ is the delay of a single pipeline stage for the optimum number of registers introduced inside of a single round. In FPGA implementations, this

delay is determined by the delay of a single CLB Slice and delays of interconnects between CLBs. As a result, the throughput does not depend on the complexity of a cipher round and tend to be similar for a large number of block ciphers, including AES. Therefore, the full mixed inner- and outer-round pipelining should be the architecture of choice for the implementations of AES targeting the highest possible throughput.

The choice of a hardware architecture depending on the resistance to side channel attacks is beyond the scope of this chapter. However, it should be noted that if the countermeasures against the side channel attacks are introduced at the circuit or logic levels, as proposed in multiple papers, such as [37–40], then all hardware architectures presented in this chapter might be equally secure.

10.10 Exercises

1. Using your knowledge about the internal structure of the *SubBytes* and *InvSubBytes* transformations, verify the correctness of the following entries of the AES S-box and AES Inverse S-box:

   a. S-box[89] = A7
   b. InverseS-box[89] = F2

2. Compute an output of the *MixColumns* transformation for the following sequence of input bytes "12 45 78 AB". Apply the *InvMixColumns* transformation to the obtained result to verify your calculations. Change the first byte of the input from "12" to "02", perform the *MixColumns* transformation again for the new input, and determine how many bits have changed in the output.
3. Compute the first two round keys of AES corresponding to the 128-bit key of all ones.
4. Draw a block diagram of the modified basic iterative architecture capable of encrypting messages in the counter mode using the minimum number of clock cycles. Compute the total time necessary to encrypt a message of the length of 1 MB using hardware implementation of Rijndael with a 128-bit input block and a 256-bit key, working in the counter mode with the size of a message block $k = 8$, assuming the modified basic iterative architecture operating with the clock frequency of 25 MHz.
5. Compute the total time necessary to encrypt a message of the length of 1 kB using hardware implementation of Rijndael with 128-bit input block and 192-bit key, working in the CFB mode with the size of a message block $k = 32$, assuming the basic iterative architecture operating with the clock frequency of 30 MHz.
6. Derive a formula for the contents of the look-up tables $T_0{}^{-1}$, $T_1{}^{-1}$, $T_2{}^{-1}$, $T_3{}^{-1}$ used for the decryption in the T-box-based implementation of AES. Compute the contents of the following components of these tables: $T_0{}^{-1}[1]$, $T_1{}^{-1}[2]$, $T_3{}^{-1}[254]$.

10.11 Projects

1. Develop and compare three different implementations of the *SubBytes InvSubBytes* operations using look-up tables, look-up tables and logic, and logic only.
   - compare the minimum clock period and use of logic resources (CLB Slices, Block RAMs, etc.) among the three designs
   - pipeline each design in order to obtain
     – minimum possible clock period and
     – maximum throughput to area ratio
     compare the three designs in terms of these parameters.
2. Develop and compare three different implementations of the *MixColumns InvMixColumns* operations using
   a. basic implementation
   b. compact implementation with parallel *InvMixColumns* decomposition
   c. compact implementation with serial *InvMixColumns* decomposition

   Compare the three designs in terms of the total resource usage, minimum latency for the *MixColumns* operation, and minimum latency for the *InvMixColumns* operation. Determine how suitable is each of the three designs for a hardware architecture with deep inner-round pipelining.
3. Develop and compare two different high-level implementations of AES in the basic iterative architecture
   a. based on S-boxes
   b. based on T-boxes

   Compare both implementations in terms of the maximum throughput, area, and throughput to area ratio.
4. Develop and compare three different implementations of the AES key scheduling unit with the number of output bits per clock cycle equal to
   a. 32 bits
   b. 64 bits
   c. 128 bits

   Compare all three implementations in terms of the minimum clock period and area.
5. Develop and compare three different implementations of the compact architecture of AES with the datapath width equal to
   a. 8 bits
   b. 32 bits
   c. 64 bits
   d. 128 bits (basic iterative architecture)

   Compare all four implementations in terms of the minimum clock period and area.

References

1. FIPS 197: Advanced Encryption Standard. National Institute of Standards and Technology, 2001, available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
2. Amphion. Documentation of cryptographic cores, available at http://www.amphion.com
3. D. Canright. A very compact Rijndael S-box. Technical Report NPS-MA-05-001, 2005.
4. D. Canright. A very compact S-box for AES. In J. R. Rao and B. Sunar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'05)*, LNCS, vol. 3659, pp. 441–455. Springer-Verlag, 2005.
5. P. Chodowiec. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. Master's thesis, George Mason University, Mar. 2002.
6. P. Chodowiec and K. Gaj. Very compact FPGA implementation of the AES algorithm. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, LNCS vol. 2779, pp. 319–333. Springer-Verlag, 2003.
7. J. Daemen and V. Rijmen. AES proposal: Rijndael. Technical Report, 1999, available at http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf
8. J. Daemen and V. Rijmen. *The design of Rijndael: AES - The Advanced Encryption Standard*. Number ISBN 3-540-42580-2. Springer-Verlag, 2002.
9. A. Dandalis, V. K. Prasanna, and J. D. Rolim. A comparative study of performance of AES final candidates using FPGAs. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems Workshop (CHES'00)*, LNCS, vol. 1965 pp. 125–140. Springer-Verlag, 2000.
10. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*, pp. 13–27. New York, USA, Apr. 13–14, 2000.
11. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on VLSI Systems*, 9(4):545–557, 2001.
12. V. Fischer. Realization of the round 2 candidates using Altera FPGA. *Comments for The Third Advanced Encryption Standard Candidate Conference (AES3)*, New York, USA Apr. 13–14, 2000.
13. V. Fischer and M. Drutarovský. Two methods of Rijndael implementation in reconfigurable hardware. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems (CHES'01)*, LNCS vol. 2162, pp. 81–96. Springer-Verlag, 2001.
14. V. Fischer, M. Drutarovský, P. Chodowiec, and F. Gramain. InvMixColumn decomposition and multilevel resource sharing in Rijndael implementation. *IEEE Transactions on VLSI Systems*, 13(8):989–992, 2005.

15. V. Fischer and F. Gramain. Resource sharing in a Rijndael implementation based on a new MixColumn and InvMixColumn relation. unpublished.
16. K. Gaj and P. Chodowiec. Hardware performance of the AES finalists-survey and analysis results. Technical Report, George Mason University, 2000, available at http://ece.gmu.edu/crypto/AES_survey.pdf
17. K. Gaj and P. Chodowiec. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*, pp. 40–54. New York, USA, Apr. 13–14, 2000.
18. K. Gaj and P. Chodowiec. Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays. In *Proc. The Cryptographer's Track at the RSA Security Conference (CT-RSA'01)*, LNCS vol. 2020, pp. 84–99. Springer-Verlag, 2001.
19. T. Good and M. Benaissa. AES FPGA from the fastest to the smallest. In J. R. Rao and B. Sunar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'05)*, LNCS, vol. 3659, pp. 427–440. Springer-Verlag, 2005.
20. Helion. Documentation of cryptographic cores. Available at http://www.heliontech.com
21. T. Ichikawa, T. Kasuya, and M. Matsui. Hardware evaluation of the AES finalists. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*, pp. 279–285. New York, USA, Apr. 13–14, 2000.
22. K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2003)*, pp. 207–215. Monterey, CA, Feb. 23–25, 2003.
23. H. Kuo and I. Verbauwhede. Architectural optimization for a 1.82Gbits/sec VLSI implementation of the AES Rijndael algorithm. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems (CHES'01)*, LNCS, vol. 2162, pp. 51–64. Springer-Verlag, 2001.
24. I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Trans. Computer-Aided Design*, 62(2), Feb. 2007.
25. A. Lutz, J. Treichler, F. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fichtner. 2Gbit/s hardware realizations of RIJNDAEL and SERPENT: A comparative analysis. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, LNCS, vol. 2523, pp. 144–158. Springer-Verlag, 2002.
26. U. Mayer, C. Oelsner, and T. Köhler. Evaluation of different Rijndael implementations for high end servers. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, vol. 2, pp. 348–351. Scottsdale, Arizona, USA 2002.
27. S. McMillan and C. Patterson. JBits implementations of the Advanced Encryption Standard (Rijndael). In *Proc. Field-Programmable Logic and Applications (FPL'01)*, LNCS, vol. 2147, pp. 162–171. Springer-Verlag, 2001.

28. N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. A systematic evaluation of compact hardware implementations for the Rijndael S-box. In J. R. Rao and B. Sunar, editors, *Proc. (CT-RSA'05)*, LNCS, vol. 3376, pp. 323–333. Springer-Verlag, 2005.
29. S. Morioka and A. Satoh. A 10 Gbps full-AES crypto design with a twisted-BDD S-Box architecture. In *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 98–103. Freiburg, Germany, 2002.
30. S. Morioka and A. Satoh. An optimized S-Box circuit architecture for low power AES design. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, LNCS, vol. 2523, pp. 172–186. Springer-Verlag, 2002.
31. A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. Rao, and P. Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems (CHES'01)*, LNCS, vol. 2162, pp. 171–184. Springer-Verlag, 2001.
32. G. Saggese, A. Mazzeo, N. Mazzocca, and A. Strollo. An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In *Proc. International Conference on Field-Programmable Logic and Applications (FPL'03)*, LNCS, vol. 2778, pp. 292–302. Springer-Verlag, 2003.
33. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A compact Rijndael hardware architecture with S-Box optimization. In *Proc. Theory and Application of Cryptology and Information Security (ASIACRYPT'01)*, LNCS, vol. 2248, pp. 239–254. Springer-Verlag, 2001.
34. P. R. Schaumont, H. Kuo, and I. M. Verbauwhede. Unlocking the design secrets of a 2.29 Gb/s Rijndael processor. In *Proc. ACM Conference on Design Automation (DAC 2002)*, pages 634–639. New Orleans, Louisiana, USA, 2002.
35. N. Sklavos and O. Koufopavlou. Architectures and VLSI implementations of the AES-Proposal Rijndael. *IEEE Transactions on Computers*, 51(12): 1454–1459, 2002.
36. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat. Efficient implementation of Rijndael encryption in reconfigurable hardware: improvements and design tradeoffs. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, LNCS, vol. 2779, pp. 334–350. Springer-Verlag, 2003.
37. K. Tiri, D. Hwang, A. Hodjat, B. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. Prototype IC with WDDL and differential routing - DPA resistance assessment. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, LNCS, vol. 3659, pp. 354–365. Springer-Verlag, 2005.
38. K. Tiri and I. Verbauwhede. Securing encryption algorithms against dpa at the logic level: next generation smart card technology. In Ç. K. Koç, C. Paar, and C. D. Walter, editors, *Cryptographic Hardware and Embedded*

Systems - CHES 2003, LNCS, vol. 2779, pp. 125–136, Cologne, Germany, 2003. Springer-Verlag.

39. K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proc. of Design Automation and Test in Europe (DATE 2004)*, pp. 246–251. Paris, France, 2004.
40. K. Tiri and I. Verbauwhede. A VLSI design flow for secure side-channel attack resistant ICs. In *Proc. of Design Automation and Test in Europe (DATE 2005)*, pp. 58–63, 2005.
41. I. Verbauwhede, P. Schaumont, and H. Kuo. Design and performance testing of a 2.29-GB/s Rijndael processor. *IEEE Journal of Solid-State Circuits*, 38(3):569–572, 2003.
42. B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware performance simulations of Round 2 Advanced Encryption Standard algorithms. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*. New York, USA, Apr. 13–14, 2000.
43. J. Wolkerstorfer. An ASIC implementation of the AES MixColumn operation. In *Proc. Austrochip 2001*, pp. 129–132. Vienna, Austria, Oct. 12, 2001.
44. J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES SBoxes. In *Proc. The Cryptographer's Track at the RSA Security Conference (CT-RSA 2002)*, LNCS, vol. 2271, pp. 67–78. Springer-Verlag, 2002.
45. A. C. Zigiotto and R. d'Amore. A low-cost FPGA implementation of the Advanced Encryption Standard algorithm. In *Proc. Symposium on Integrated Circuits and Systems Design (SBCCI'02)*, pp. 191–196. Porto Alegre, Brazil, 2002.

Chapter 11
Secure and Efficient Implementation of Symmetric Encryption Schemes using FPGAs

François-Xavier Standaert

11.1 Introduction

Due to its potential to greatly accelerate a wide variety of applications, reconfigurable computing has gained importance in the industrial development of digital signal processing systems. This chapter discusses how the particular properties of field programmable gate arrays (FPGAs) can be exploited for the secure and efficient implementation of symmetric cryptographic algorithms and protocols.

Reconfigurable computing intends to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. Reconfigurable devices such as FPGAs contain arrays of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, sometimes known as logic blocks, are connected using a set of routing resources that are also programmable. As a consequence, the realization of FPGA designs can be performed at the user site. Synthesis and implementation tools allow the high-level description of a hardware design to be translated into the programming file for an FPGA. The run-time operation of a reconfigurable system consequently occurs in two distinct phases: configuration and execution. First, the programming file of the reconfigurable device is directed from a host PC or an on-board memory to the FPGA. This configuration data are used to define the actual operation of the hardware. Thereafter, during the execution phase, the reconfigurable device acts as a purpose-built hardware.

The structure of actual computation blocks within the reconfigurable hardware varies from system to system. Each computation unit, or logic block, can be as simple as a 3-input function generator (usually denoted as look-up table (LUT)), or as complex as an 8-bit arithmetic and logic unit (ALU). This difference in the block size is commonly referred to as the granularity of the logic block. Fine-grain

UCL Crypto Group
e-mail: fstandae@uclouvain.be

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_11,

© Springer Science+Business Media, LLC 2009

blocks are useful for bit-level manipulations while coarse-grain blocks are better optimized for high-level data manipulations. The granularity of the FPGA also has a potential impact on the configuration time of the device. A fine-grained array has many configuration points to perform very small computations, and thus requires more data bits during reconfiguration. Recent FPGAs such as the one illustrated in Figure 11.1 usually combine different sizes or types of blocks in order to efficiently support different kinds of computations. For example, standard logic blocks using 4-input LUTs are combined with embedded RAM blocks, multipliers and microprocessors. Next to the computational blocks, the interconnections also have a major impact in the final performance of an FPGA. Recent devices are usually structured in different lengths of interconnects in order to efficiently deal with close and remote connections between the different logic blocks.

In the remainder of this chapter, we assume a reader with basic knowledge in FPGA design and cryptographic algorithms. Rather than providing a general introduction to reconfigurable cryptographic implementations, this chapter aims to put forward a number of properties of these devices and to discuss how they can be exploited efficiently and securely. Underlining how FPGA designs differ from standard integrated circuit designs with this respect is an alternative goal. Due to the very general nature of this topic, it is not intended to be extensively covered and our different sections attempt (as far as possible) to redirect the reader toward further readings when necessary. As introduction to the following issues, we suggest [10] for a general report on reconfigurable computing, [22] for detailed descriptions

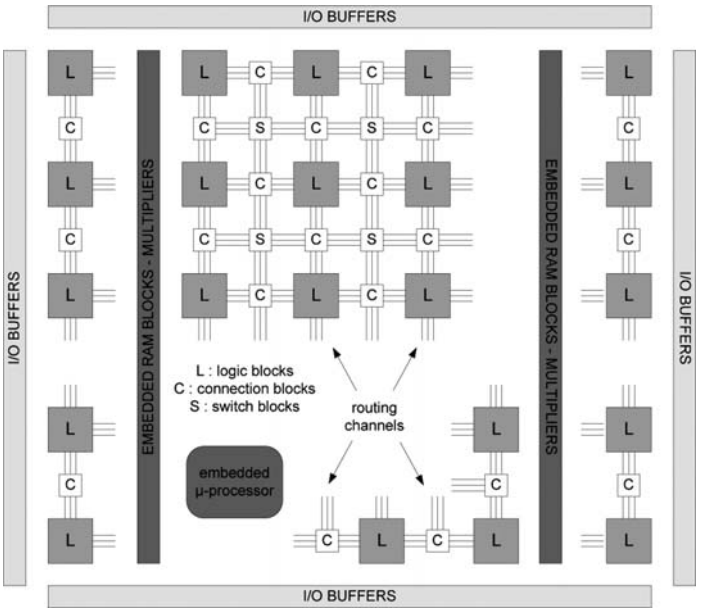

Fig. 11.1 High-level view of an FPGA.

on the efficient implementation of cryptographic algorithms on FPGAs, [11] for a good bibliography on FPGA security issues and [36] for a combined discussion of implementations and attacks on FPGAs. Additionally, since most of our running examples are using Xilinx FPGAs, we refer to [37] for a detailed description of these devices.

The rest of the chapter is structured as follows. Section 11.2 discusses the efficient exploitation of different FPGA features, from low-level facilities of the logic blocks to high-level architectural features. Section 11.3 details some metrics used for evaluating the efficiency of a cryptographic implementation. The two last sections investigate different issues related to the security of reconfigurable devices. Section 11.4 comments on the applicability of an important class of physical attacks (denoted as side-channel attacks) to recent FPGAs. Section 11.5 surveys other security topics, related to fault insertion and bitstream security. Our conclusions are in Section 11.6. We note finally that most of the examples used in this chapter are borrowed from implementation works of the UCL Crypto Group. Many similar results can be found in the literature and this chapter does not aim to give an overview of previously published works.

11.2 Efficient FPGA Implementations

This section considers the exploitation of different features in recent reconfigurable devices for the efficient implementation of symmetric cryptographic algorithms. We first investigate the slice structure, then describe the exploitation of additional embedded blocks and conclude by discussing the possible advantages of higher-level architectural facilities provided by recent FPGAs, namely embedded microprocessors and dynamic reconfiguration. We note that most of these features directly derive from the FPGA datasheets. However, since they are not always optimally (and automatically) exploited by the synthesis and implementation tools, it is important to have them in mind already during the high-level description of a cryptographic design.

11.2.1 Exploiting the Slice Structure

In this first section, we consider a Xilinx Virtex-II FPGA. Such devices embed programmable logic blocks, RAMs and multipliers. The slice is the logic unit that is generally used to evaluate an FPGA design's area requirements. Such a slice, depicted in Figure 11.3, is made up of two LUTs, two flip-flops (or registers) and a few additional gates. According to the user's choice, any of these LUTs can be configured in one out of three possible ways: RAM16 that acts like a $2^4 \times 1$-bit RAM storage, SRL16 that implements a 16-bit linear shift register and LUT that is capable of computing any 4-to-1 boolean function. A more detailed view of half a slice is given in Figure 11.2. An interesting thing to notice is the fast carry chain

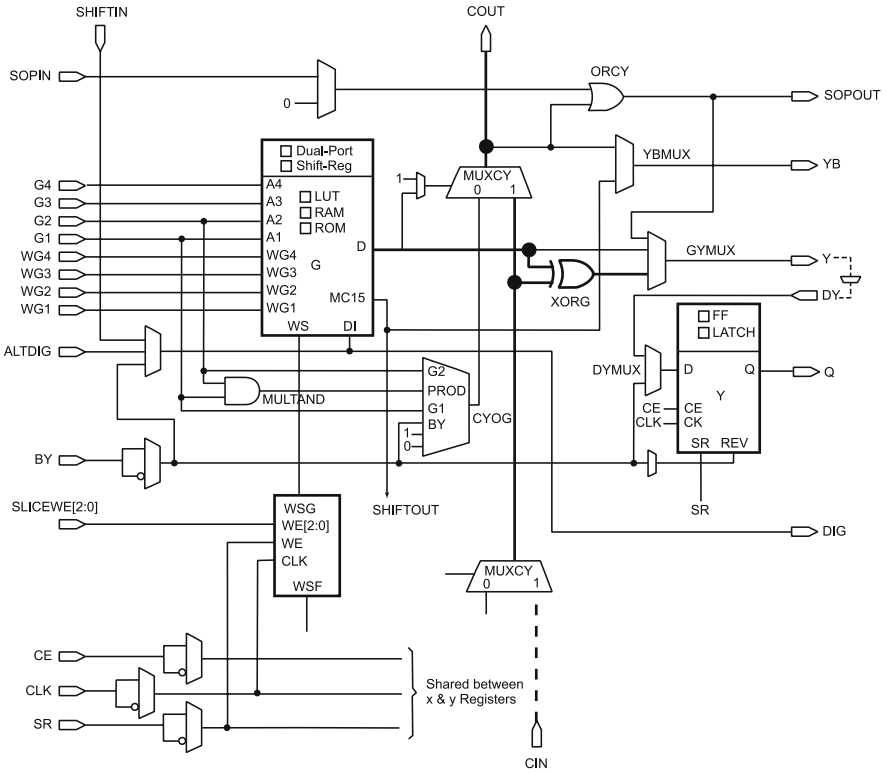

Fig. 11.2 Top half slice of a Xilinx Virtex-II.

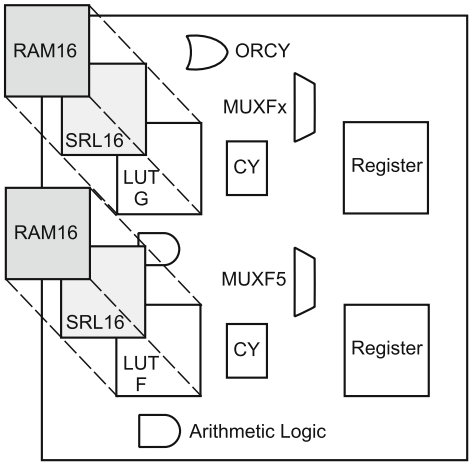

Fig. 11.3 Slice configurations.

(emphasized on the picture) crossing the slice. It allows the efficient implementation of carry propagate adders and, as will be shown, can sometimes be used to simplify the combinatorial cost of some designs.

11.2.1.1 The "Maximum" Pipeline Strategy and Limitations

The first observation from these pictures is that any LUT in a slice comes with its flip-flop. A consequence is that a straightforward pipelining strategy can be applied to the hardware design of, e.g., block ciphers in which the number of pipeline stages equals (or is close to) the design's logic depth expressed in LUTs. This is in contrast with ASICs where every flip-flop has its cost. Such strategies, looking for the maximum pipeline, generally give rise to very good synthesis results. However, the implementation (especially the routing task) of block ciphers within certain FPGAs can then become the bottleneck, due to high data diffusion. As mentioned, e.g., in [28], the design of a maximum pipelined advanced encryption standard (AES) Rijndael exhibits delays with 20% of logic and 80% of routes. It suggests that such a strategy is not optimum in these contexts. Improved solutions involve either the use of registers to pipeline the routes (if very high frequencies are to be reached) or the limitation of the design logic depth to two (or more) LUTs (if high efficiencies are required).

11.2.1.2 The Slice Multiplexors

Next to the slice LUTs, Xilinx FPGAs provide multiplexors (usually denoted as multiplexors Fx) allowing to efficiently implement distributed RAM and ROM within the FPGA. These elements have a strong impact on the implementation efficiency of encryption algorithms using substitution boxes, e.g., the AES Rijndael that uses a $2^8 \times 8$ S-box. As an illustration, the previously described Virtex-II allows implementing a $2^8 \times 1$ ROM with 16 LUTs and consequently, the AES Rijndael S-box fits in 64 slices. In the recent Virtex-5 family of FPGAs, the 4-input LUTs have been turned into 6-input LUTs. Using the same additional multiplexors allows to implement a $2^8 \times 1$ ROM with only 4 LUTs and consequently the AES Rijndael S-box fits into 32 slices.

11.2.1.3 The Shift-Register Slice Structure

A second convenient feature of the Virtex-II slice is the possibility to configure the LUT as a 16-bit shift register. This feature has a significant impact, e.g., in the implementation of stream ciphers using linear feedback shift registers. It is worth noticing that the efficient exploitation of the SRL16 primitives highly depends on the required taps positions in the stream ciphers. Any time an intermediate value is extracted from the SRL16 primitives, a new LUT has to be used. As a consequence, it may happen that a k-bit register with $k < 16$ occupies a complete SRL16 cell.

The stream cipher grain implementation in [7] is a good example of a straightforward (but very efficient) exploitation of these shift register-configured LUTs, with convenient taps positions.

Another interesting use of the shift register structure occurs when the target ciphers have an unbalanced structure. As a typical example, the data encryption standard (DES) has a very light key scheduling algorithm, compared to its round function. Therefore, the maximum pipeline strategy applied to the round and key round results in different number of pipeline stages. In a maximum pipeline implementation with "on-the-fly" round key derivation, several slices will consequently be "wasted" to pipeline the key schedule (meaning that their corresponding LUT will not be used). In such a context, the shift register structure can provide up to 16 pipeline stages with one single LUT, which result in a much more efficient implementation, e.g., in [23].

11.2.1.4 Additional Logic Gates Within the Slice

Finally, configurable logic blocks generally embed additional logic gates that can be efficiently exploited in certain specific contexts. One classical example is the XOR gate that is illustrated by the emphasized path in Figure 11.2. Since most symmetric encryption algorithms make an extensive use of such gates, they are generally useful to cryptographic designers. As an illustration, combining this gate with one LUT allows implementing a 5-bit XOR operation. In a maximum pipeline AES Rijndael implementation taking advantage of this 5-bit XOR, the combination of the MixColumns and AddRoundKey operations can consequently fit in only two pipeline stages, e.g., in [28]. Note that there is generally only one such XOR gate for several LUTs in a slice, which has to be taken into account during the designing phase (e.g., two LUTs can share the same XOR gate, but it has to have the same input).

11.2.2 Exploiting Embedded Blocks

Most recent FPGAs have an hybrid structure combining fine-grain logic blocks with larger-grain, specialized embedded blocks. Next to the inner structure of the FPGA logic blocks that was previously discussed, this section investigates how these larger blocks can be useful in the context of symmetric cryptographic implementations. For illustration, we selected two frequently available such blocks, namely embedded memories and multipliers.

11.2.2.1 RAM Blocks

Just as distributed RAM and ROM can implement the S-boxes of a block cipher, embedded memories can play the same role. However, in order to efficiently exploit these blocks, it is important to fill them as completely as possible, e.g., with the

substitution tables defined in the target cipher specifications. As an illustration, the RAM blocks in the Virtex-E devices are dual-ported 4096-bit synchronous. Since the AES S-box has $2^8 \times 8 = 2048$ bits of memory requirements, it means that two S-boxes can fit in one such block. By contrast, Virtex-II devices incorporate dual-port synchronous RAM blocks of 18 Kbit. Storing the Rijndael S-boxes in such blocks is consequently not an efficient solution. An alternative proposal to exploit these larger memories is to implement both the S-boxes and the MixColumns operation as precomputed tables. Namely, let us consider the combination of SubBytes and MixColumns in Rijndael. An output column of this transform equals

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} SB(a_0) \\ SB(a_1) \\ SB(a_2) \\ SB(a_3) \end{bmatrix},$$

where the b_i's represent the combined transform output bytes and the a_i's its input bytes to the S-boxes. Therefore, if we define four tables as

$$T_0(a) = \begin{bmatrix} 02 \times SB(a) \\ SB(a) \\ SB(a) \\ 03 \times SB(a) \end{bmatrix}, \quad T_1(a) = \begin{bmatrix} 03 \times SB(a) \\ 02 \times SB(a) \\ SB(a) \\ SB(a) \end{bmatrix},$$
$$T_2(a) = \begin{bmatrix} SB(a) \\ 03 \times SB(a) \\ 02 \times SB(a) \\ SB(a) \end{bmatrix}, \quad T_3(a) = \begin{bmatrix} SB(a) \\ SB(a) \\ 03 \times SB(a) \\ 02 \times SB(a) \end{bmatrix},$$

the combination of SubBytes and MixColumns equals

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = T_0(a) \oplus T_1(a) \oplus T_2(a) \oplus T_3(a).$$

The size of one T_i table is $2^8 \times 32 \simeq 8$ Kbits. It is consequently possible to store the four tables in two dual-port 18 Kbit RAM blocks, e.g., as in [24]. Similarly, such 18 Kbit memory blocks offer a straightforward solution to implement a masked DES design to improve the security against side-channel attacks (see Section 4.2 for details). As the DES S-boxes have $2^6 \times 4$ bits of memory requirements, its masked counterpart has $2^{12} \times 4 \simeq 16$ Kbit of memory requirements, which just fits into the Virtex-II RAM blocks [32]. It is finally important to note that due to their fixed position within the device, embedded RAM blocks impose stronger routing constraints than their distributed counterpart of the previous section. It may affect the operating frequency of a design, specially if a large number of embedded blocks are used.

11.2.2.2 Embedded Multipliers

Although arithmetic operands are more likely to be exploited in asymmetric cryptographic applications, there are examples of symmetric ciphers in which one can also

take advantage of embedded multipliers. In the IDEA block cipher, integer multiplication modulo $(2^n + 1)$ is usually the bottleneck for hardware implementations. Such constraints suit pretty well to FPGAs embedding small multiplier blocks, such as the Virtex-II family of devices [4].

11.2.3 Exploiting Further Features

The previously described FPGA features related to the logic block structure and the use of embedded memories and multipliers have been intensively used in a variety of implementation works. In this section, we briefly describe some more recent (and less-investigated) trends in the design of reconfigurable systems that can potentially be exploited in cryptographic applications.

11.2.3.1 Microprocessors and Controllers

Microprocessors and controllers of two shapes can be found in recent reconfigurable devices. First, hard cores can be discretely embedded in the device, as the previously described RAM blocks and multipliers. This is typically the case of the PowerPC microprocessor that is available in recent Xilinx devices, e.g., the Virtex-II-pro. Second, micro-controllers can be synthesized and implemented within the FPGA logic blocks, just as distributed memories. This is typically the case of the MicroBlaze (32-bit RISC) and PicoBlaze (8-bit RISC) controllers that are freely available as soft cores from Xilinx.[1] In a processor-based FPGA system, customized IP cores can then be connected to the controllers through various interfaces like the on-chip-peripheral bus (OPB) or the fast simplex link (FSL). In general, processor-based embedded systems cannot be compared favorably with specialized cryptographic designs in which the hardware is optimized and possibly pipelined in order to reach high implementation efficiencies. However, when the system specifications establish that a general-purpose processor must be used, the MicroBlaze and PicoBlaze solutions can be suitable. They additionally offer the flexibility of being programmed with a software language. For example, [13] describes the implementation of various block ciphers within a MicroBlaze-based system.

11.2.3.2 Dynamic Reconfiguration

Dynamic reconfiguration refers to the possibility of reconfiguring an FPGA partially, while operating and without compromising the integrity of the application

[1] As an illustration, a PicoBlaze core takes less than 200 logic cells in a Spartan-II device and can run at 76 MHz. A MicroBlaze core takes less than 1000 logic cells in a Virtex-II device and can run at 125 MHz.

running. It is sometimes referred to as partial or run-time reconfiguration [18]. From a theoretical point of view, dynamic reconfiguration allows using more hardware than physically present in the FPGA which can be used to reduce the size of the device as well as its overall power consumption. From an application point of view, the expected benefits include any adaptive change of the FPGA design due to environmental changes (e.g., a change of algorithm or change of performance constraints). However, the exploitation of such techniques pose a number of practical issues, including the reliability of the design flow and the time required for the reconfiguration. Its exploitation in cryptographic applications is therefore a scope for further research, although it appears as a promising opportunity to improve systems flexibility.

11.2.4 Combining the Tricks: The Flexibility Versus Efficiency Tradeoff

To conclude this section, let us first mention that all the previous tricks only constitute a part of the possibilities offered by recent FPGAs, can be efficiently combined and generally have to be considered during the high-level modeling stage of a hardware design. For example, the way the inner structure of the slice can be exploited strongly determines the pipelining strategy to use. Second, it is important to consider that the optimal exploitation of one specific target FPGA, by designing in function of the slice structure or available embedded blocks, makes the hardware code less portable. It also sometimes requires to map some parts of the design by hand into the FPGA resources. There is consequently a tradeoff to find between the efficient exploitation of a given device and the possibility to use an IP core in a variety of systems and products. Note finally that although our illustrative implementation examples are based on symmetric cryptographic algorithms, the techniques discussed in this section generally apply for any reconfigurable hardware design.

11.3 Fair Evaluation of a Cryptographic FPGA Design

Before any cryptographic design is implemented always comes the question of the performance goals to achieve. Stating these goals properly in function of a target application and determining good metrics for the performance evaluation is therefore an important step in the understanding of reconfigurable architectures. Unfortunately, there is no straightforward answer to these questions and the fair evaluation (or comparison) of a given FPGA implementation is often a matter of taste. This section aims to illustrate some important questions to consider in such a performance assessment. For this purpose, we restrict ourselves to the implementation of the AES Rijndael. We start by considering the design goals. Then we discuss the performance evaluation.

11.3.1 Design Goals

A list of design goals for the FPGA implementation of the AES Rijndael would typically include (but is not limited to) the following eight questions:

1. Does the application require to develop an encryption/decryption core or just an encryption only, decryption only core?
2. Is the key scheduling algorithm required to be performed "on-the-fly" or can the round keys be computed once and stored in memory?
3. Is the block cipher design supposed to run in a specific encryption mode (e.g., feedback) that would prevent the use of pipelining?
4. What kind of interface has to be provided to the outside world?
5. Are there specific constraints to be fulfilled by the implementation (e.g., in terms of hardware cost or throughput)?
6. What is the target FPGA device? With which speedgrade?
7. Are there available embedded blocks in the device (are not they required for running other applications than cryptographic ones)?
8. What is the datapath size planned for the design (128-bit, 32-bit, etc.)?
9. Are multiple clocks allowed within the reconfigurable system?

11.3.2 Performance Evaluation

Assuming a hardware designer has implemented the AES Rijndael following some of the design goals in the previous section, its performances could then be measured with the following metrics: hardware cost (in LUTs, registers, slices, etc.), operating frequency (in MHz), throughput (in Mbit/sec) and possibly some efficiency measurement, e.g., throughput/hardware cost. For illustration, Table 11.1 lists some exemplary AES Rijndael implementations with selected design goals and Table 11.2 summarizes their performances according to selected metrics. These tables typically illustrate the difficulty of performing fair comparisons between different FPGA designs. First, different architectures generally have different design goals. Second, evaluation metrics can be misleading since they highly depend on the target device. Comparing the performances of different algorithms raises similar questions.

Table 11.1 Exemplary AES Rijndael designs with selected design goals.

Index	E,D	Key Sched.	Feedback	Device	Architecture
1.	E only	On-the-fly	no	Virtex-E	128-bit unrolled
2.	E only	On-the-fly	no	Virtex-E	128-bit loop
3.	E/D	Precomputed	yes	Virtex-II	32-bit loop
4.	E/D	Precomputed	yes	Spartan-II	8-bit loop
5.	E/D	Precomputed	yes	Spartan-II	PicoBlaze

Table 11.2 Exemplary AES Rijndael designs with selected performance metrics.

Index	Ref.	LUTs	Regs.	Slices	RAMBs	Freq.	Throughput
1.	[28]	3516	3840	2784	100	92 MHz	11.7 Gbit/sec
2.	[28]	3846	2517	2257	0	169 MHz	2 Gbit/sec
3.	[24]	288	113	146	3	123 MHz	358 Mbit/sec
4.	[14]	–	–	124	2	67 MHz	2.2 Mbit/sec
5.	[14]	–	–	119	2	90 MHz	710 Kbit/sec

Note that efficiency metrics (e.g., throughput/hardware cost) can be specially misleading since the hardware cost in FPGAs can be expressed in LUTs, slices, RAMBs, etc. Some metrics consequently attempt to unify these different resources, e.g., by expressing the cost of the RAM blocks as distributed RAMS in LUTs, but this is still device dependent. General observations can nevertheless be highlighted. For example, looking at the dependencies between the architecture size and the throughput in the previous tables, one could state that applications in the multi-Gbit/sec range should consider 128-bit unrolled architectures, applications in the Gbit/sec range should consider 128-bit loop architectures, applications in the hundreds of Mbit/sec range should consider 32-bit loop architectures and so on. As previously mentioned, these tables are far from being a complete survey of existing implementations of the AES Rijndael nor do they contain the best available results. For a more detailed list of such implementations, we refer to [15].

11.4 Security of FPGAs Against Side-Channel Attacks

The previous sections mainly cared about efficient FPGA implementations. However, as far as cryptographic algorithms are concerned, not only their hardware cost, throughput, etc. are important to a designer but also their security against various types of physical attacks. Physical attacks on cryptographic devices take advantage of implementation-specific characteristics to recover the secret parameters involved in the computations. They are therefore much less general – since specific to a given implementation – but often much more powerful than classical cryptanalysis and are considered very seriously by cryptographic device manufacturers. Examples of physical attacks include the probing of devices [2], the insertion of faults [5] or the monitoring of side-channel information leakages such as the power consumption [17] or electromagnetic radiation [1]. Due to the important amount of public work that has been dedicated to the analysis of side-channel attacks against FPGAs, this section discusses their specificities. In Section 11.5, we consider other aspects related to the tamper resistance of reconfigurable devices, including fault attacks and bitstream security issues. As for the previously discussed efficiency concerns, we do not aim to present an exhaustive survey of physical attacks on FPGAs but to put forward a number of their meaningful features. We redirect the reader toward further readings when needed.

11.4.1 Applicability of the Attack and FPGA Properties

Side-channel attacks are based on the hypothesis that an exploitable amount of secret information is leaked by an implementation through a physical channel. For example, in power analysis attacks, an attacker uses a hypothetical model of the device under attack to predict its power consumption. These predictions are then compared to the real measured power consumption in order to recover secret information (i.e., secret key bits of block ciphers). In this first section, we aim to illustrate that such physical information is indeed leaked by FPGA devices and can be exploited, using simple attack models. For this purposes, we focus on static RAM-based reconfigurable devices (like the previously considered Xilinx Virtex family) since they are the most popular technology in use. In these devices, the storage cells, the logic blocks and the connection blocks are made of CMOS gates.

11.4.1.1 A Simple Leakage Model Applicable to FPGAs

Static CMOS gates have three distinct dissipation sources [21]. The first one is due to the leakage currents in transistors. The second one is due to the so-called short-circuit currents there exists a short period during the switching of a gate while NMOS and PMOS transistors are conducting simultaneously. Finally, the dynamic power consumption is due to the charge and discharge of the load capacitance C_L represented by the dotted paths in Figure 11.4. The respective importance of these dissipation sources typically depends on technology scalings. But the dynamic power consumption is particularly relevant from a side-channel point of view since it determines a simple relationship between a device's internal data and its externally observable power consumption. It can be written as

$$P_{dyn} = C_L V_{DD}^2 P_{0\to 1} f, \tag{11.1}$$

where $P_{0\to 1}$ is the probability of a $0 \to 1$ bit transition, f is the operating frequency of the device and V_{DD} is the voltage of the power supply. Therefore, in practice, a

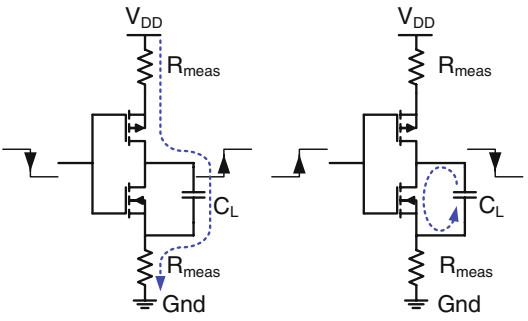

Fig. 11.4 Charge versus discharge of a CMOS inverter.

simple way to model the power consumption of an FPGA is to predict its switching activity. Let S_1 be a large bit register containing the state of all the FPGA cells at some moment in time t_1 and S_2 be another register containing the FPGA state one clock cycle later. The number of bit switches (including both $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions) in the device when moving from state S_1 to S_2 equals the Hamming distance between these states, namely $H_D(S_1, S_2) = H_W(S_1 \oplus S_2)$, where H_W is the Hamming weight operator.

11.4.1.2 Exploiting the Leakages

We illustrate the attack principle with the simple encryption network of Figure 11.5, which contains the same basic elements as most present block ciphers e.g., the AES Rijndael. That is, the plaintext is XORed with a secret key, then goes through a layer of relatively small substitution boxes and is finally sent to a larger permutation (e.g., a linear diffusion layer for the AES Rijndael). The same operations are iterated a number of times. For the purposes of this chapter, it is not necessary to know more details on these algorithms. The attack proceeds as follows. Let the adversary target the 4 key bits entering the left S-box of Figure 11.1, denoted as $K_0[0...3]$. Then, for N different plaintexts, he first predicts the number of transitions at the S-box output, for every possible value of $K_0[0...3]$. The result of this prediction is a $N \times 2^4$ prediction matrix $\mathbf{P}$, containing numbers between 0 and 4. In the second part of the attack, the adversary lets the circuit encrypt the same N plaintexts with a fixed secret key and he measures the power consumption of the device while the chip is operating the targeted operation. For each plaintext, he stores a single value for the power consumption (e.g., the average or maximum value of the target clock cycle). This results in a $N \times 1$ measurement vector $\mathbf{M}$. Finally, the attacker computes the correlation[2] between the measurement vector and all the columns of the prediction

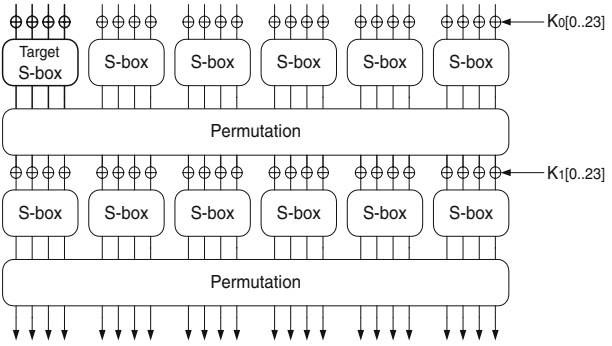

Fig. 11.5 A simple encryption network.

[2]

$$C(\mathbf{M}, \mathbf{P}) = \frac{\mu(\mathbf{M}.\mathbf{P}) - \mu(\mathbf{M}).\mu(\mathbf{P})}{\sqrt{\sigma^2(\mathbf{M}).\sigma^2(\mathbf{P})}}, \tag{11.2}$$

where $\mu(\mathbf{M})$ denotes the mean of the set of measurements and $\sigma^2(\mathbf{M})$ its variance.

matrix (corresponding to all the possible key guesses). If the attack is successful, it is expected that only one value, corresponding to the correct key bits, leads to a high correlation.

Such attacks have been successfully applied to different algorithms implemented on a variety of FPGA devices. For example, an attack against the simple design of Figure 11.5 has been implemented against a Xilinx Spartan-II device and its results are illustrated in Figure 11.6 in which the correct key candidate is clearly distinguishable. We note that different statistical tools could be considered to mount power analysis attacks and the use of the correlation coefficient is not optimal with this respect. For example, maximum likelihood techniques [9] may yield better results. However, with the simple power consumption models considered here, correlation attacks provide good results and are extremely easy to manipulate (e.g., they do not require any estimation of the noise in the target devices). Note finally that the same set of measurements can be used to recover all parts of the key, by changing the prediction matrix (i.e., by applying a divide and conquer strategy).

11.4.1.3 Exemplary FPGA Properties

To summarize the previous paragraphs, there are two important aspects to take into account in the analysis of side-channel attacks. First, the target implementation has to leak some information. With this respect, recent FPGAs are made of CMOS

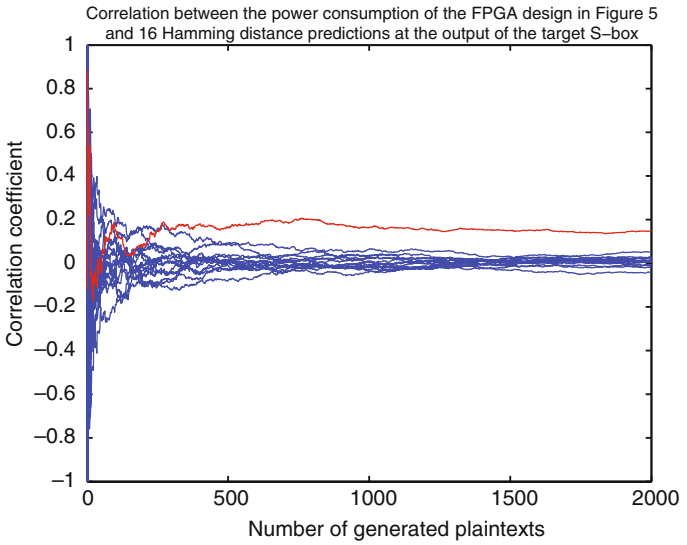

Fig. 11.6 Exploitation of the side-channel leakages.

transistors, just as smart cards and controllers. They consequently leak information, just as smart cards and controllers. Second, this information has to be exploitable by the adversary. With this respect, FPGA implementations offer opportunities to perform parallel computing and are based on a specific array of logic and routing cells. These features (among others) lead to specific approaches in the exploitation of side-channel leakages.

- *Parallel computing and target leakage*. In power analysis attacks, the leakage provides an adversary with an image of the computation performed within a target device. Depending on the implementation context (e.g., see the different architectures in Table 11.1), this information relates to 8-bit, 32-bit, 128-bit (or more) computations. But looking at Figure 11.5, a side-channel adversary typically targets small parts of the computation one by one, corresponding to small (e.g., 4-bit) parts of the key. Therefore, the power consumption due to the untargeted parts of the computation generates what is usually denoted as algorithmic noise. Compared to smart cards and controllers, FPGAs offer the specific opportunity to design large architectures, with a significant amount of such noise.
- *FPGA structures and leakage models.* Even more specific of FPGAs is the array structure of Figure 11.1. In this structure, the different computational elements are connected through different types of wires. A consequence is that the different bits in an implementation contribute differently to the overall power consumption, due to different effective capacitances. These different capacitances have been highlighted, e.g., in [26] for the Virtex-II family of devices. A consequence is that the simple Hamming distance model for the prediction of the power consumption in reconfigurable devices can be improved according to these effective capacitances, e.g., by assigning different weights to the switches of different bits within a design. As an illustration, the left part of Figure 11.7 depicts

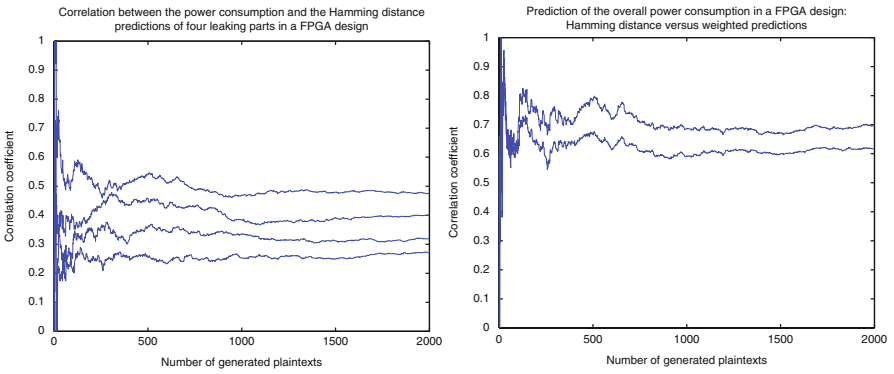
Correlation between the power consumption and the Hamming distance predictions of four leaking parts in a FPGA design
Correlation coefficient
Number of generated plaintexts
Prediction of the overall power consumption in a FPGA design: Hamming distance versus weighted predictions
Correlation coefficient
Number of generated plaintexts

Fig. 11.7 Resource-dependent correlations in Xilinx FPGAs.

the correlation of four parts of a computation in an FPGA and their contribution to the overall power consumption (measured with a correlation coefficient). The right part of the figure illustrates that assigning different weights to these computations allows a better prediction of the overall power consumption [31], which results in a more efficient side-channel attack.

To conclude this section, the state-of-the-art side-channel attacks against FPGA devices, e.g., as surveyed in [30], typically illustrate the following three facts: (1) Side-channel are a threat for FPGAs, as for any microelectronic cryptographic device. (2) Just as FPGAs offer nice opportunities for efficient implementations, they have interesting features for secure implementations as well (e.g., the parallel computing opportunity or the ability to design datapaths including the countermeasures of the following section). (3) Just as the optimal exploitation of the FPGA structure is useful for efficiency (as detailed in Section 11.2), exploiting the architectural properties of a given target device (e.g., the effective capacitance of the different computational parts in a design) is useful for improving the efficiency of an attack.

11.4.2 Countermeasures

Countermeasures against side-channel attacks range among a large variety of solutions. However, in the present state of the art, no single technique allows to provide perfect security. Protecting implementations against physical attacks consequently intends to make the adversary's task harder. In this context, the implementation cost of a countermeasure is of primary importance and must be evaluated with respect to the additional security obtained. The exhaustive list of all possible solutions to protect cryptographic devices from side-channel opponents would deserve a long survey in itself. In the following, we list four illustrative solutions to improve the resistance against power analysis that are applicable to FPGAs. Obtaining practical security usually requires to combine them (possibly with others) in a clever way.

- *Noise addition.* Adding noise to the side-channel measurements is a very common technique to reduce the amount of information in the leakages. This can be achieved in a variety of ways, at different abstraction levels, e.g., physical, technological, algorithmic. As previously mentioned, the use of large architectures producing a significant amount of algorithmic noise is interesting with this respect and easy to apply to FPGA designs [29].
- *Data randomizations* intend to make all the cryptographic computations within the FPGA dependent on some unknown random values generated on-chip. It makes the prediction of the power consumption more difficult. Masking is a typical example of such countermeasures that has been intensively studied in the literature and applied to FPGAs, e.g., in [20, 32].
- *Random pre-charges* are another solution to make the side-channel leakage harder to exploit. If one every two inputs to an encryption design is a random number generated on chip, an adversary will not be able to predict the transitions

within the implementation anymore (of course, the random ciphertexts should not be outputted from the device). As suggested in [31], a solution for the adversary is then to distinguish between $0 \rightarrow 1$ and $1 \rightarrow 0$ bit transitions in the leakages. But it results in worse leakage models and less-efficient attacks than in the un-protected Hamming distance model.

- *Dynamic and differential logic styles* finally intend to make the power consumption within the FPGA independent of the computed data. A logic style is denoted as differential if the complementary data inputs and outputs are available in the circuit. The notion of dynamic logic gates refers to the fact that the gate operation is divided into two phases [21]. First, the output capacitance is charged. Then, during the evaluation, it is discharged according to the input values. When combining dynamic and differential logic styles, there are always two capacitances loaded during the pre-charge and one of them is discharged during the evaluation, regardless of the input sequences. In [34], such a circuit behavior is proposed for FPGAs.

11.4.3 Measuring Side-Channel Resistance

Countermeasures against side-channel attacks as listed in the previous section usually involve a significant performance overhead for the encryption algorithms. Therefore, just as hardware efficiency is a design goal that has to be evaluated with (hopefully) fair metrics, physical security also has to be evaluated properly. In this section, we briefly refer to the proposed evaluation methodology introduced in [33] for these purposes. We use the intuitive picture of Figure 11.8 in which side-channel attacks are viewed as a communication problem. In summary, there are two important aspects to consider in the analysis of side-channel attacks. First, the amount of information leaked by a target device can be measured with the conditional entropy (or mutual information). Second, the extent to which an adversary can exploit this information can be measured with its success rate, just as the bit-error-rate does in

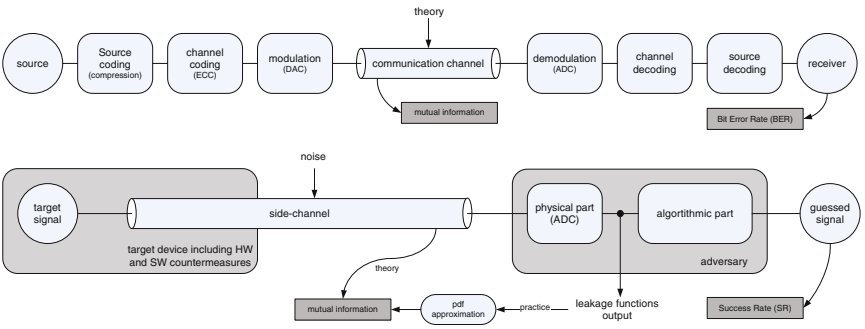

Fig. 11.8 Digital communications versus side-channel attacks.

communication problems. By combining both measurements, one can evaluate both the quality of an implementation and the strength of a side-channel adversary. Trading efficiency for security consequently requires to evaluate how the addition of a countermeasure in a design affects these two metrics.

11.5 Other Security Issues

Side-channels attacks are only a part of the concerns related to the implementation of cryptographic algorithms within reconfigurable devices. *Readback attacks, cloning of the devices, bitstream reverse-engineering and fault attacks* are other concerns that could be considered [36]. In this section, we review some issues related to the insertion of faults in FPGAs and to their bitstream security. They both constitute research challenges for future applications.

11.5.1 Fault Attacks

Similarly to side-channel attacks, fault attacks are an intensively studied adversarial model for cryptographic implementations. However, only a small number of experiments can be found in the literature on the actual possibility to apply such attacks to FPGAs. Preliminary results as in [8] suggest that fault insertion is feasible (as for any other SRAM-based device), but could be more difficult to exploit than in the context of smaller devices like smart cards. The large number of memory cells present in the logic arrays, determining at the same time the computational state of a cryptographic algorithm and the configuration of the device (including logic and routing), implies that different types of faults can occur. "How efficiently can these faults be exploited?" or "Can they hurt the FPGAs permanently?" are exemplary open questions. As for side-channel attacks, it is reasonable to expect that security against such attacks will require to add countermeasures (some of them surveyed in [19] for block ciphers) and therefore to trade some of a design's efficiency for security.

11.5.2 Bitstream Security

Bitstream security is a critical issue for SRAM-based FPGAs. The recovery of bitstreams (e.g., by applying readback attacks in which the configuration file is read out of the FPGA) in order to clone an FPGA or their reverse-engineering and also the digital rights management (DRM) of the intellectual property (IP) cores are important concerns for the electronic industry. In this section, we survey some of these questions and describe the (partial) solutions that have been proposed by the industry. We start with a (simplified) description of the different parties in the game. Then, we discuss the interactions between these parties and the related security problems.

11.5.2.1 Parties in the FPGA Business

In order to keep our descriptions as simple and straightforward as possible, we limit our discussions to a three-player game, namely the "*end user*", the "*system designer*" and the "*IP provider*". A more detailed description of the FPGA IP transactions can be found in [16]. The IP provider delivers the hardware description language (HDL) files (or any other suitable file format such as the netlist for a particular FPGA) for some specific algorithms, e.g., encryption, image processing. The system designer creates a complete design for an FPGA chip, making use of one or more IP cores purchased from IP providers, e.g., a hardware decoder for the digital cinema [25]. Finally, the end user takes advantage of equipment containing FPGAs.

11.5.2.2 Bitstream Security: System Designer Versus End User

In this interaction, the main goal for the system designer is to prevent readback attacks, cloning of the FPGAs and reverse-engineering of the bitstream. Otherwise said, the FPGA should appear as black box to the end user. Since the bitstream is generally stored in an EPROM, an additional issue is to securely connect this external memory and the FPGA. For all these purposes, the most frequently considered solution is the bitstream encryption illustrated in Figure 11.9. In this solution, the bitstream is encrypted by the CAD tool with user-defined symmetric (secret) keys. The same keys are stored on the FPGA, e.g., in a volatile memory with an external battery. During configuration, an on-chip decryption circuit is used to recover the original configuration file. Readback is not allowed when encrypted bitstreams are used.

Although this or similar methods are used in several commercial devices, they suffer from a number of drawbacks. First, it requires an external battery to store the key. Second, it requires an on-chip decryption circuitry. But most importantly,

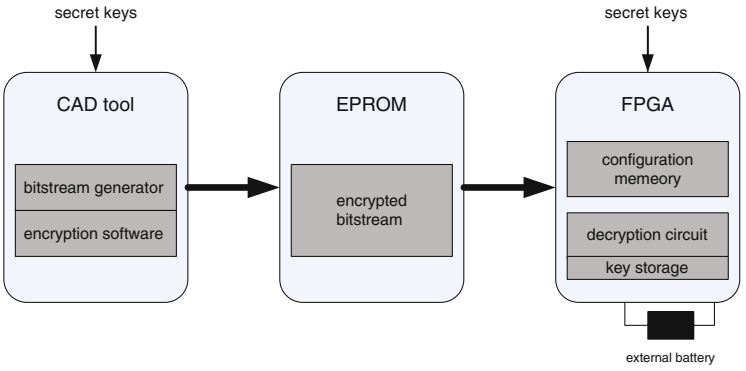

Fig. 11.9 Bitstream encryption in Virtex-II FPGAs.

the key management of such a solution is tricky. For example, if a single key is used for all the boards, then a system designer has no opportunity to update the configuration files for only a part of them. Ideally, it should be possible to update the symmetric keys remotely. This could be achieved either by the use of a symmetric master key (but the system security would then depend on this single key) or a public key mechanism in which each FPGA would come with a private/public key pair stored in a non-volatile memory.

11.5.2.3 IP Cores DRM: IP Provider Versus System Designer

In this interaction, the situation is even more difficult. First, the IP provider does not have the FPGA devices in hand which prevents him to store symmetric keys as in the previous section. Second, the system designer would like to be able to integrate the IPs in a larger design and to simulate it.

In present devices, a solution for the IP provider to deal with these issues is to send the system designer an encrypted netlist and simulation model. Xilinx development tools allow dealing with such files by embedding a secret key in its software. But the security of this solution entirely relies on this single key which may consequently be the target of reverse-engineering attempts. In addition, this model only allows a "per IP core license" business model.

A proposal to allow a "per device license" business model is described in [38]. For this purpose, and for every unit to be built, the IP provider feeds the system designer with both an IP core and a pre-programmed external security chip. A secret key (chosen by the IP provider) is stored in the encrypted netlist of the IP and the same key is embedded in the security chip. Before the IP can run on a board, it checks that the security chip embedding the correct key is properly connected to the FPGA. The hardware to do this security check is part of the IP core. This solution allows the IP provider to monitor the number of devices running its design but suffers from several drawbacks. First, the key management is not easy since the key is embedded in the netlist. As for the bitfile encryption, key updates could be made easier by using either a master key or a public key mechanism in the FPGAs. Second, the system security does still rely on netlist encryption in the Xilinx software. If an adversary can decrypt the netlist, he can also disable the security check.

There is consequently a tradeoff to face in the DRM of IP cores. For flexibility reasons, it is desirable that the security relates to the netlists so that IPs can be easily simulated and integrated in larger designs (as in the previous proposal). But for security reasons, the best solution would be to deal directly with bitstreams. For example, if a non-volatile private key K_s and the corresponding public key K_p were available in an FPGA, an IP provider could sell a pair $[E_K(\text{bitstream}), E_{K_p}(K)]$ to protect its design, in a "per device license" business model. But present development tools do not allow to easily combine different bitstreams which makes this solution quite unpractical for the system designer's point of view. Improved solutions (e.g.,

taking advantage of partial reconfiguration techniques) are consequently required to improve this setting. Note finally that, whatever the DRM and bitstream security mechanisms involved, the underlying cryptographic algorithms may still be the target of side-channel or fault attacks. It is therefore important to quantify the reliability of these solutions with an appropriate security level.

11.6 Conclusions and Open Questions

This chapter discussed some aspects in the secure and efficient implementation of symmetric encryption schemes in recent FPGAs. It aims to illustrate both how the particular properties of these reconfigurable devices can be exploited to improve the performances of an implementation and how the same properties can be exploited by malicious adversaries. Our discussions suggest different tradeoffs for cryptographic designers. First, the flexibility of a design can be traded for performances. That is, by carefully taking advantage of all the architectural details of a given device, one can improve performance at the cost of a less-portable hardware code. Second, the performances of a design can be traded for physical security. That is, resisting against fault or side-channel attacks usually involves overheads in the design efficiency.

From a technological point of view, open questions in the field relate to the effect of technology scalings in the future generations of FPGAs, both in terms of performances and security against physical adversaries. From an application point of view, and as the capacity of FPGAs increases to millions of equivalent gates, the protection of IP cores with secure DRM solutions becomes increasingly important. The development of IP protection schemes that do not harm the flexibility of the development tools is therefore an important requirement. It should allow IP providers, FPGA system designers and end users to interact in a fair and secure business model. The integration of a public key mechanism by FPGA manufacturers or the exploitation of physically unclonable functions within FPGAs, e.g., as suggested in [27, 35] appear as promising approaches with this respect.

Acknowledgements The author would like to thank Fraçois Macé, Guerric Meurice and Gaël Rouvroy for meaningful comments on this work.

11.7 Exercises

Khazad [3] is an iterated 64-bit block cipher with 128-bit keys. It comprises eight rounds; each round consists of eight 8-bit to 8-bit S-box parallel look-ups, a linear transformation (multiplication by a constant MDS diffusion matrix) and round key addition. For efficient hardware implementations, the 8-bit to 8-bit substitution is made of six smaller 4-bit to 4-bit substitutions.

1. Assume an FPGA with 4-bit LUTs and dual-ported 4096-bit synchronous RAM blocks. What is the cost of the complete Khazad substitution layer in LUTs and RAMBs? What are the respective memory requirements (in bits) of the LUT-based and RAMB-based solutions for the S-box?
2. Consider the 64-bit loop architecture for a (simplified) round of Khazad in Figure 11.7. Assume that each layer (NL1, NL2, NL3, L) has a cost of 64 LUTs (fully utilized). What is the total cost of such a design in LUTs (including the key addition and the input multiplexor)?
3. Assume a very simple key scheduling that can be implemented as a single layer of 128 LUTs. How much LUTs and registers are required to pipeline such a key scheduling in five levels in the best manner if a slice structure similar to the one of Figure 11.3 is used?
4. Assume that the delay of a LUT equals 5 nsec and that only these delays determine the operating frequency of the design. What maximum throughput can be obtained with a 2 (*resp*. 5) pipeline stage strategy if eight rounds have to be iterated (in Mbits/sec)?
5. Same question if the delay of a LUT equals 3 nsec but there is a fixed delay due to routing constraints in the design of 10 nsec. What is the best throughput that can be obtained in a feedback mode in this context?
6. Consider a side-channel adversary trying to recover a n-bit random value k who obtains the Hamming weight of this value: $H_W(k)$. Assuming noiseless measurements, how much information does he gain? Now assume $n = 64$ as for the Khazad cipher and an adversary who would obtain the Hamming weight of first round S-box layer's output for different plaintexts. Can an adversary exploit all the information on k in a correlation attack? (hint: think both about information and computation).

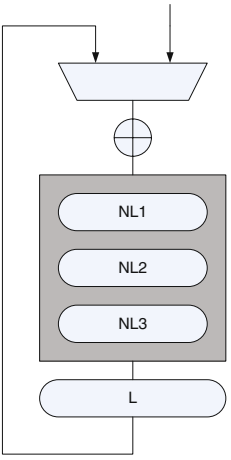

Fig. 11.10 Loop architecture for a (simplified) round of Khazad.

7. Consider a side-channel adversary targeting part of the key in the design of Figure 11.7. Assume that the NL layers do not provide cryptographic diffusion and that the L layer does provide perfect diffusion. Assume a five pipeline stage implementation in which the overall power consumption of the design is only caused by the registers. Assume that the correlation between the predictions of the adversary and the real measurements (denoted as ρ) equals the square root of the number of predicted registers in an attack divided by the total number of registers in the design. Assume finally that the number of plaintexts required for a successful attack can be approximated with $N_{succ} \simeq c \cdot \frac{1}{\rho^2}$. How many plaintexts are required for a successful attack against an 8-bit part of the key for which $c = 10$?
8. Consider now an implementation in which two rounds of Khazad are unrolled. How much would the security against the previous side-channel adversary be increased? Consider finally the same implementation protected by a countermeasure such that the correlation coefficient is reduced by a factor of 5. What number of plaintexts would then be required to attack?
9. Evaluate the hardware cost and throughput of the previous two-round unrolled implementation of Khazad ($\Delta_{\text{lut}} = 5$ nsec). Then assume that the previous countermeasure against side-channel attacks (reducing the correlation coefficient by a factor of 5) is applied to a single round architecture, divides the throughput by two and uses 250 additional LUTs in the design (e.g., it could be a design with random pre-charges). Can you comment the efficiency versus security against correlation attacks tradeoff for these designs? Which metrics can be used for these purposes?

11.8 Projects

Select a target symmetric cryptographic algorithm and a list of design goals, e.g., from Section 11.3.1.

1. Design a reconfigurable hardware architecture for this algorithm, simulate it and implement it for a target device from any FPGA manufacturer.
2. Evaluate the resulting efficiency of your design according to the metrics of Section 11.3.2.
3. Compare it with the existing literature and put forward the weaknesses of your design.
4. What specific features of your FPGA did you exploit?
5. Then, think about hardware security from a general point of view. Select a physical attack that you want to prevent and add one or several countermeasure(s) to your design.
6. Comment on the efficiency versus security tradeoff.
7. Does your countermeasure add new physical weaknesses?

References

1. D. Agrawal, B. Archambeault, J. Rao, and P. Rohatgi. *The EM Side-Channel(s), in the proceedings of CHES 2002*, LNCS, vol. 2523, pp. 29–45, Redwood City, California, USA, August 2002.
2. R. Anderson and M. Kuhn. *Tamper Resistance – a Cautionary Note, in the proceedings of the USENIX 1996*, pp. 1–11, Oakland, USA, November 1996.
3. P. Barreto and V. Rijmen. *The KHAZAD Legacy-Level Block Cipher*, available from: http://www.cosic.esat.kuleuven.ac.be/nessie/
4. J. L. Beuchat. Modular multiplication for FPGA implementation of the IDEA block cipher, Research Report, num 2002-32, ENS Lyon, September 2002.
5. D. Boneh, R. DeMillo, and R. Lipton. *On the Importance of Checking Cryptographic Protocols for Faults, in the proceedings of Eurocrypt 1997*, LNCS, vol. 1233, pp. 37–51, Konstanz, Germany, May 1997.
6. L. Bossuet, G. Gogniat, and W. Burleston. *Dynamically Configurable Security for SRAM FPGA Bitsreams, in the proceedings of IPDPS 2004*, pp. 146–158, Los Alamitos, CA, USA, April 2004.
7. P. Bulens, K. Kallach, F.-X. Standaert, and J.-J. Quisquater. *FPGA Implementation of eSTREAM Phase-2 Focus Candidates with Hardware Profile, in the proceedings of SASC 2007*, Bochum, Germany, February 2007.
8. V. Maingot, J. B. Ferron, G. Canivet, and R. Leveugle. Fault attacks on SRAM-based FPGAs, USEIT Security Workshop, Toulouse, France, July 2007.
9. S. Chari, J. Rao, and P. Rohatgi. *Template Attacks, in the proceedings of CHES 2002*, LNCS, vol. 2523, pp. 13–28, Redwood City, CA, USA, August 2002.
10. K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, v. 34(2): 171–210, June 2002.
11. S. Drimer. FPGA design security bibliography webpage, http://www.cl.cam.ac.uk/∼sd410/fpgasec/
12. S. Drimer. *FPGA Bitstream Authentication: Why and How*, in the proceedings of ARC 2007, LNCS, vol. 4419, pp. 73–84, Rio de Janeiro, Brazil, March 2007.
13. I. Gonzales and F. J. Gomez-Arribas. Ciphering algorithms in microBlaze-based embedded systems. *IEE Proceedings, Computers and Digital Technologies*, 153(2): 87–92, March 2006.
14. T. Good and M. Benaissa. *AES on FPGA: From the Fastest to the Smallest, in the proceedings of CHES 2005*, LNCS, vol. 3659, pp. 427–440, Edinburgh, UK, September 2005.
15. K. Jarvinen, M. Tommiska, and J. Skytta. Comparative survey of high-performance cryptographic algorithm implementations on FPGAs. *IEE Proceedings*, 152(1): 3–12, October 2005.
16. T. Kean. *Cryptographic Rights Management of FPGA IP Cores, in the proceedings of FPGA 2002*, pp. 113–118, Monterey, CA, USA, February 2002.
17. P. Kocher, J. Jaffe, and B. Jun. *Differential Power Analysis*, in the proceedings of Crypto 1999, LNCS, vol. 1666, pp. 398–412, Santa-Barbara, USA, August 1999.

18. P. Lysaght, B. Blodget, J. Young, and B. Bridgford. *Enhanced Architectures, Design Methodologies And CAD Tools For Dynamic Reconfiguration of Xilinx FPGAs, in the proceedings of FPL 2006*, Madrid, Spain, September 2006.
19. T. G. Malkin, F.-X. Standaert, and M. Yung. *A Comparative Cost/Security Analysis of Fault Attack Countermeasures, in the proceedings of FDTC 2005*, LNCS, vol. 4236, pp. 159–172, Edinburgh, Scotland, September 2005.
20. E. Peeters, F.-X. Standaert, N. Donckers, and J.-J. Quisquater. *Improved Higher-Order Side-Channel Attacks With FPGA Experiments, in the proceedings of CHES 2005*, LNCS, vol. 3659, pp. 309–323, Edinburgh, Scotland, September 2005.
21. Jan M. Rabaey. Digital Integrated Circuits, Prentice Hall International, 1996.
22. F. Rodriguez, N. A. Saqib, A. D. Perez, and Ç. K. Koç. *Cryptographic Algorithms on Reconfigurable Hardware*, Springer, 2006.
23. G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat. *Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES, in the proceedings of FPL 2003*, LNCS, vol. 2778, pp. 181–193, Lisbon, Portugal, September 2003.
24. G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat. *Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications, in the proceedings of ITCC 2004*, Las Vegas, USA, April 2004.
25. G. Rouvroy, F.-X. Standaert, F. Lefebvre, and J.-J. Quisquater. *Reconfigurable Hardware Solutions for the Digital Rights Management of Digital Cinema, in the proceedings of DRM 2004*, pp. 40–53, Washington DC, USA, October 2004.
26. L. Shang, A. Kaviani, and K. Bathala. *Dynamic Power Consumption in Virtex-2 FPGA Family, in the proceedings of FPGA 2002*, pp. 157–164, Monterey, California, USA, February 2002.
27. E. Simpson and P. Schaumont. *Offline Hardware/Software Authentication for Reconfigurable Platforms, in the proceedings of CHES 2006*, LNCS, vol. 4249, pp. 311–323, Yokohama, Japan, October 2006.
28. F.-X. Standaert, G. Rouvroy, J.-D. Legat, and J.-J. Quisquater. *Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs, in the proceedings of CHES 2003*, LNCS, vol. 2779, pp. 334–350, Cologne, Germany, September 2003.
29. F.-X. Standaert, S. B. Ors, and B. Preneel. *Power Analysis of an FPGA Implementation of Rijndael: Is Pipelining a DPA Countermeasure?, in the proceedings of CHES 2004*, LNCS, vol. 3156, pp. 30–44, Cambridge, MA, USA, August 2004.
30. F.-X. Standaert, E. Peeters, G. Rouvroy, and J.-J. Quisquater. *An Overview of Power Analysis Attacks Against Field Programmable Gate Arrays*, in the Proceedings of the IEEE, 94(2):383–394, February 2006.
31. F.-X. Standaert, E. Peeters, F. Mac, and J.-J. Quisquater. *Updates on the Security of FPGAs Against Power Analysis Attacks, in the proceedings of ARC 2006*, LNCS, vol. 3985, pp. 335–346, Delft, The Netherlands, March 2006.

32. F.-X. Standaert, G. Rouvroy, and J.-J. Quisquater. *FPGA Implementations of the DES and Triple-DES Masked Against Power Analysis Attacks, in the proceedings of FPL 2006*, Madrid, Spain, August 2006.
33. F.-X. Standaert, T. G. Malkin, and M. Yung. A formal practice-oriented model for the analysis of side-channel attacks, Cryptology ePrint Archive, Report 2006/139, 2006, available from http://eprint.iacr.org/2006/139
34. K. Tiri and I. Verbauwheder. *Synthesis of Secure FPGA Implementations, in the proceedings of the International Workshop on Logic and Synthesis (IWLS 2004)*, pp. 224–231, June 2004.
35. P. Tuyls, G. J. Schrijen, B. Skoric, J. van Geloven, N. Verhaegh, and R. Wolters. *Read-Proof Hardware from Protective Coatings, in the proceedings of CHES 2006*, LNCS, vol. 4249, pp. 369–383, Yokohama, Japan, October 2006.
36. T. Wollinger, J. Guarjardo, and C. Paar. Security on FPGAs: State of the art implementations and attacks. *ACM Transactions in Embedded Computing Systems*, 3(3):534–574, August 2004.
37. Xilinx. Virtex, Virtex-E, Virtex-II, Virte-II Pro, Virtex-4, Virtex-5 Field programmable gate arrays data sheets, http://www.xilinx.com
38. Xilinx. Xilinx FPGA identification friend of foe copy protection with 1-Wire SHA-1 secure memories, Application Note 3826, http://www.xilinx.com

Chapter 12
Block Cipher Modes of Operation from a Hardware Implementation Perspective

Debrup Chakraborty and Francisco Rodríguez-Henríquez

12.1 Introduction

Block ciphers are one of the most important primitives in cryptology. They are based on well-understood mathematical and cryptographic principles. Due to their inherent efficiency, these ciphers are used in many kinds of applications which require bulk encryption at high speed.

Generally speaking, a block cipher consists of at least two closely related algorithms: block encryption and block decryption. Block encryption takes as an input a fixed-length block (known as the *plaintext*) and transforms it into another block of the same length (known as the *ciphertext*) under the action of a fixed secret key that may or may not have the same length of the plaintext. A block cipher must be invertible in the sense that by using the block decryption algorithm it should be always possible to recover the original plaintext from the ciphertext and the secret key. Figure 12.1 shows schematically the situation just described. We stress that once the plaintext has been encrypted using a given key, then a successful decryption can only be performed by knowing that key. Due to this feature, block ciphers are classified as a secret or symmetric key primitives.

Formally a block cipher is considered to be secure if it behaves like a strong pseudorandom permutation, i.e., a block cipher is secure if an adversary cannot distinguish its output from a randomly chosen permutation. This definition of security for block ciphers is very strong, it implies that for any possible input, a secure block cipher should produce random outputs. It is unfortunate that there exists no formal security model for assessing whether a block cipher is or not secure or rather, how secure a cipher is. Hence, we rely on a block cipher by the fact that no one has been able to find an attack on it.

Assuming that an adversary has managed to obtain a plaintext and the corresponding cipher text, then she can always try to obtain the n-bit secret key of a

CINVESTAV IPN, Mexico
e-mail: debrup,francisco@cs.cinvestav.mx

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_12,

© Springer Science+Business Media, LLC 2009

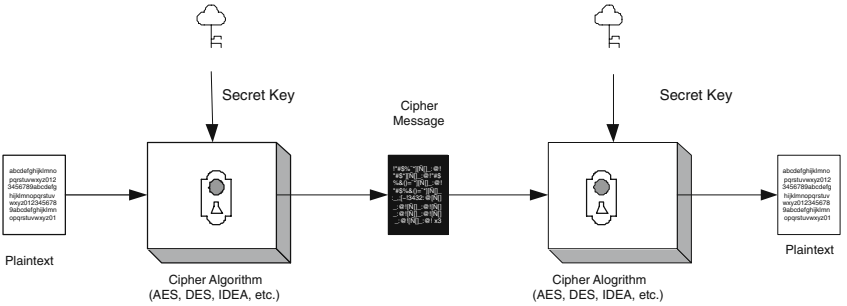

Fig. 12.1 Encrypting/decrypting with block ciphers.

given symmetric block cipher by trying all possible keys, a procedure traditionally termed *brute force attack*. We say that a block cipher has a security strength of n bits if the best known attack against it is not computationally cheaper than the brute force attack. Modern block ciphers typically use a block and key length ranging from 64 bits up to 256 bits. At the present state of the technology, block/key sizes of 128 bits are generally considered adequate in terms of both security and efficiency.

Block ciphers have been around for civilian/commercial use since 1971, when a team leaded by H. Feistel and his colleagues at IBM designed a family of ciphers known as Lucifer [2, 53]. Early versions of Lucifer operated on 24-bit long plaintext blocks. The strongest variant which was released in 1973, operated on 128-bit blocks and 128-bit secret keys. A revised version of that Lucifer variant, known as the data encryption standard (DES), was adopted as a US FIPS standard in 1974 [16, 42]. Across the years, DES became the most influential block cipher ever inspiring many new designs and attacks. Some other examples of famous block ciphers include IDEA, AES, RC6, etc.

Most block ciphers have an iterative design which implies that the block being encrypted/decrypted is processed by repeatedly applying a simpler function called *round*. Typically, the number of rounds in modern ciphers ranges from 10 up to 32. It is also customary to use a different sub-key per round, which is sometimes called *round key*. Round keys are usually derived from the user secret key mentioned before, through a process called *key schedule*. Hence, a contemporary block cipher specification usually comprises three different algorithms, namely, encryption, decryption and key schedule algorithms.

As we have seen, block ciphers can only process plaintexts/ciphertexts with a bit length smaller than blocklength of the block cipher, which is typically less than 256 bits. But this is an unacceptable restriction since applications demand encryption/decryption of arbitrary long messages. In order to overcome this difficulty, it is necessary to introduce the concept of a *mode of operation*, which we will define next.

A block cipher can be viewed as a function $E : \mathscr{K} \times \mathscr{M} \rightarrow \mathscr{C}$, where $\mathscr{K} \in \{0,1\}^k$ and $\mathscr{M}, \mathscr{C} \in \{0,1\}^n$. Then, a mode of operation can be defined as a procedure that takes as input a key $K \in \{0,1\}^k$, a message $P \in \{0,1\}^*$ of arbitrary length and

sometimes an initialization vector or *nonce* $IV \in \{0,1\}^v$ and produces a ciphertext $C \in \{0,1\}^*$ as its output. During the encryption process, some modes also produce a tag $\tau \in \{0,1\}^l$ with $0 < l \leq n$ that can be considered as a checksum or hash value of the plaintext message.[1] The notion of a tag value is useful for offering the security service of data integrity/authentication.

More informally, a mode of operation is a specific way to use a block cipher to enable it to encrypt arbitrary long messages and (optionally) to provide other security services, such as data confidentiality/privacy, authentication or a combination of both.

Let us now assume for a moment that we have a secure block cipher which produces outputs that are indistinguishable from random strings. Unfortunately, even if we manage to obtain such a strong cipher, it is not guaranteed that we can use it securely to encrypt arbitrary long messages. To illustrate this point, let us introduce next the most naive (and arguably the most insecure) mode of operation: electronic code book (ECB).

Let us consider an arbitrary plaintext message P of bitlength l. Then, we can partition the plaintext P into $b = \lceil l/n \rceil$ plaintext blocks $P_1, P_2, \ldots, P_b$ of length n, where n is the block length handled by the cipher. It is noticed that if the message length l is not a multiple of n, then the last plaintext block would be incomplete, but for the sake of simplicity, let us assume that l is a multiple of n. Then, the l-bit ciphertext C can be produced by invoking the block cipher a total of b times, thus producing b cipher blocks C_i given as $C_i = E(K, P_i)$ for $i = 1, 2, \ldots, b$. The procedure just outlined is known as the electronic code book (ECB) mode of operation.

ECB is highly insecure when dealing with plaintexts that exhibit high symmetry at the block level. For instance, Figure. 12.2a shows a 256×256 byte chess board in grayscale. If we use a block cipher with block length of 128 bits (such as AES), then the corresponding ECB-encrypted image will keep the same symmetry of the plaintext (see Figure 12.2b). This shows that designing a scheme able to encrypt arbitrary long messages using a given block cipher is not trivial.

The earliest modes of operation reported in the open literature were described back in 1981, in the standard FIPS Pub. 81 [17].[2] In that document four modes of operation were specified, namely, the electronic code book (ECB), cipher block chaining (CBC), cipher feedback (CFB) and output feedback (OFB) modes, where the data encryption standard (DES) was the underlying block cipher.

Likewise, FIPS Pub. 46-3 [16, 42] approved the seven modes specified in ANSI X9.52 [1]. Four of those modes were equivalent to the ECB, CBC, CFB and OFB modes with the triple DES algorithm (TDEA) as the underlying block cipher, whereas the other three modes in ANSI X9.52 were variants of the CBC, CFB and OFB modes. In [46], the counter mode of operation was added to the list of approved modes of operation.

[1] These modes of operation termed "*authenticated encryption modes*" are discussed in detail in Section 12.6.3.

[2] In fact, counter mode encryption ("CTR mode") was already introduced by Diffie and Hellman in 1979 [14, 32].

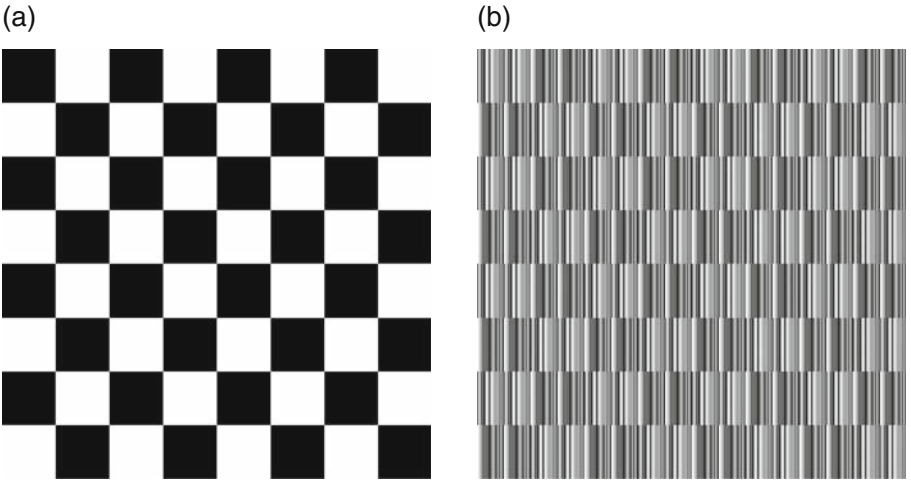

Fig. 12.2 (**a**) A 256×256 byte chess board. (**b**) The ECB-encrypted image using AES-128.

By the end of the last century, several papers pointed toward new directions on modes of operation research. The definition of the type of security provided by block ciphers was fundamental toward developing modes of operations. As stated earlier, a block cipher is considered as a pseudorandom permutation, thus the basic problem to be addressed was to find domain extensions for pseudorandom permutations. The work by Luby and Rackoff [35] was fundamental in this direction. Later, in [4] various security notions for security in the symmetric setting was presented, it also provided formal analysis of the traditional modes of operations proving security of some of them and giving the security bounds. In 2000, formal security notions of authenticity for symmetric encryption was presented in [5]. These led to numerous proposals for authenticated encryption schemes. In another development, Naor and Reingold [44] proposed a scheme to construct a strong pseudorandom permutation using a block cipher. This work along with the one reported in [33] was one of the first proposals for a class of constructions later called as tweakable enciphering schemes or disk encryption schemes.

The second generation of modes of operation, therefore, was designed to offer other security services according to different application goals. Some of the most important classes of modes of operation are those modes which guarantee confidentiality, modes for authenticated encryption, modes for authenticated encryption with associated data and modes for disk encryption.

Another characteristic that distinguishes the second generation of modes of operation is the fact that they are mostly designed for operating with several or even arbitrary selections of block ciphers. The idea that a mode of operation is a research problem largely independent of the specific block cipher being used may seem quite natural nowadays. Nevertheless, 25 years ago, when the first generation modes of operation were being specified, they were usually associated to a specific block cipher (typically DES or triple DES) [16, 17, 42].

A remarkable feature of modes of operation is the fact that, in contrast with what we have for block ciphers, a formal model for assessing their security is available.[3] Applications require various kinds/levels of securities and once a strict security model can be established for a given application, one can have a construction of a mode of operation secure under that model. Hence, a modern mode of operation is always proposed with a security bound proof valid within the model of analysis and the given security definition that provides bounds in terms of adversarial resources.

Block ciphers in different modes of operation have been implemented on all kinds of hardware and software platforms. For example, AES software implementations [7, 19] have a throughput that ranges from 300 to 800 Mbps depending on the specific architecture and platform selected by the developers. Some efficient encryptor/decryptor core VLSI implementations have also been reported in [27, 36, 51]. Performance of VLSI implementations ranges from 2 to 7.5 Gbps for the AES block cipher. Similarly, various reconfigurable hardware implementations have been reported in [8, 18, 21, 26, 34]. Those are one round (*iterative*) or n rounds (*pipeline*) FPGA implementations optimized for encryption or encryption/decryption processes. Reported performance results are broadly variable ranging from 300 Mbps to up to 25 Gbps.

Various design strategies are used for the hardware implementation of a typical block cipher and the corresponding modes of operation. An iterative looping design (IL) implements only one round and n iterations of the algorithm are carried out by feeding back previous round results. It utilizes less area (in terms of hardware resources) but consumes more clock cycles, causing a relatively low-speed encryption. In a loop unrolling or pipeline design (PP), rounds are replicated and registers are provided between the rounds to control the dataflow. The design offers high speed but area requirements are also high.

In fact, the specific selection of the mode of operation to be utilized will have a significant impact in the design of an architectural design for a block cipher. In the vast majority of block cipher hardware implementations, the electronic code book (ECB) mode of operation has been targeted. Arguably, this is because ECB is the simplest of all modes, which allows independent block encryption. Then, several blocks can be processed in parallel or pipeline strategies can be applied to increase performance.

Unfortunately, we are aware of just a handful of hardware designs of the other four traditional modes of operation, namely, CBC, CFB, OFB and the counter mode. Hardware designs implementing those modes of operation can be found in [3, 13, 18, 34, 37, 38].

The situation is even worse for the second generation of modes of operation, since many of them have never been implemented either in hardware or in software platforms. This rather deplorable situation may be caused for at least three factors. First, designing a new mode of operation has become quite a theoretical challenge in the sense that a formal proof of the alleged security of the new mode must be

[3] Usually under the assumption that the underlying block cipher is a strong pseudorandom permutation.

included in the proposal. More often than not, researchers focus all their energies on the arduous search of security bounds for their modes of operation, while the actual implementation of their proposals (including test vector generation) is neglected, ignored or, in the best case, delayed.

A second factor that may explain the lack of actual implementations for modern modes is due to efficiency reasons. Commonly, a hardware designer is only interested on producing the fastest or the most compact possible designs. Fast and ultra fast designs can only be obtained by utilizing (sub)pipeline architectures, which frequently prevent the usage of more sophisticated modern modes. On the other hand, compact designs are very often not compatible with modern modes because they typically include in their specification a number of costly building blocks (such as hash functions, field multipliers, etc.).

A third factor may be due to the fact the many of the second generation modes have been patented. This fact discourages both academicians and IT engineers to devote time to work on the hardware implementation of those modes.

The aim of this chapter is to give a brief overview of some of the most important modes of operation that have been proposed in the last few years. We first describe the traditional modes of operation, followed by a discussion of the security requirements and the adversary model used in modern modes. We describe in detail one authenticated encryption and one disk encryption mode. In order to illustrate our discussion we also provide as a case of study the design and hardware implementation description of a two pass authenticated encryption mode: AES-CCM, which also includes a brief description of the AES block cipher.

12.2 Block Ciphers

A block cipher can be viewed as a function $E : \mathscr{K} \times \mathscr{M} \rightarrow \mathscr{C}$, where $\mathscr{K} = \{0,1\}^k$ and $\mathscr{M} = \mathscr{C} = \{0,1\}^n$. Thus $\mathscr{K}$, $\mathscr{M}$ and $\mathscr{C}$ are finite nonempty sets of bit strings which are called the key space, the message space and the cipher space, respectively. The parameters n and k are called the block length and key lengths, respectively. These parameters can be different for different block ciphers. As evident from the definition, a block cipher takes as input a n-bit message (also called plaintext) and a k-bit key and produces a n-bit ciphertext. For any fixed $K \in \mathscr{K}$ we shall denote $E(K,P)$ as $E_K(P)$. For any key $K \in \mathscr{K}$ it is required that E_K is a permutation, i.e., the function $E_K : \{0,1\}^n \rightarrow \{0,1\}^n$ is a bijection. In other words for every ciphertext $C \in \{0,1\}^n$ there exist one and only one message $P \in \{0,1\}^n$ such that $E_K(P) = C$. So E_K being a permutation, it ensures that for every K a given E_K will have a inverse function which is generally called as $E_K^{-1}()$ or $D_K()$. Thus, D_K also maps $\{0,1\}^n$ to $\{0,1\}^n$ and $D_K(E_K(P)) = P$ and $E_K(D_K(C)) = C$ for all P and C in $\{0,1\}^n$.

The function $E_K()$ and $D_K()$ must be such that they can be easily computed and these functions should be normally publicly available. To use a block cipher, a key is randomly selected from the key space and agreed upon by the sender and receiver. This key should be kept secret.

Modern day block ciphers are usually composed of several identical transforms, denoted as rounds. In each round the plaintext or the semi-transformed plaintext gets transformed with the action of a round key, which is derived from the secret key by some specific transform steps. A concrete instantiation of a block cipher which is widely used is called the Advanced Encryption Standard (AES). To continue the modes of operation discussion we shall not depend on any specific block cipher, as a mode of operation is generally designed irrespective of the block cipher and any secure block cipher can be plugged into it. For the sake of completeness, we give a detailed description of the AES algorithm in the following section. Furthermore, a brief background information on binary extension fields is also given after that; this information will be useful when we discuss the offset codebook mode and its implementation aspects of the OCB and the AES rounds.

12.3 Introduction to AES

Rijndael block cipher algorithm was chosen in October 2000 by NIST as the new Advanced Encryption Standard (AES) [28]. In the rest of this section we shall give a brief summary of the AES encryption process.

The basic structure of AES consists of a message input (128 bits), a secret user key (128 bits) and a cipher message (128 bits) as the output. The AES cipher treats the input 128-bit block as a group of 16 bytes organized in a 4×4 matrix called the *state* matrix.

As is shown in Figure 12.3, the AES encryption algorithm consists of an initial transformation, followed by a main loop where nine iterations called *rounds* are executed. Each *round transformation* is composed of a sequence of four transformations, namely byte substitution (BS), ShiftRows (SR), MixColumns (MC) and AddRoundKey (ARK). For each round of the main loop, a round key is derived from the original key through a process called *Key Scheduling*. Finally, a last round consisting of three transformations BS, SR and ARK are executed.

The AES decryption algorithm operates similarly by applying the inverse of all the transformations described above in reverse order. In the rest of this section we briefly describe the AES round transformations, whereas the round-key derivation process will be explained in Section 12.3.5.

Rounds 1–9 consist of the application of the four basic steps to the state matrix. The order of the AES steps for these rounds are BS, SR, MC and ARK.

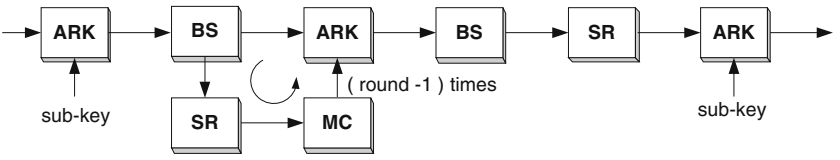

Fig. 12.3 AES encryption process.

12.3.1 Byte Substitution (BS) Step

This is the first step for rounds from 1 to 10 and it is the main non-linear transformation of the encryption process. In BS step, each input byte of the state matrix is independently replaced by another byte from a look-up table called S-box. The S-box of the AES algorithm consists of 256 entries each of one byte, where each byte is represented in the binary extension field $GF(2^8)$ constructed using the irreducible pentanomial $P(x) = x^8 + x^4 + x^3 + x + 1$. An AES S-box is composed of two transformations: First, each input byte is replaced with its multiplicative inverse in $GF(2^8)$ with the element 00 being mapped onto itself; then, an affine transformation over GF(2) is applied. The affine transformation consists of a matrix multiplication by a constant matrix followed by the addition of the hexadecimal value "63". For decryption, the inverse S-box is applied by obtaining the inverse affine transformation followed by multiplicative inversion in $GF(2^8)$.

12.3.2 Shift Rows (SR) Step

It is the second step in the round transformation, consisting of a cyclic shift operation where each row in the state matrix is rotated cyclically to the left using $0-, 1-, 2-$ and $3-$ byte offset. In decryption, the rotation is applied to the right.

12.3.3 Mix Columns (MC) Step

In MC step, each column of the state matrix, considered as a polynomial over $GF(2^8)$, is multiplied by a fixed polynomial $c(x)$ modulo $x^4 + 1$. The polynomial $c(x)$ is given by $c(x) = 03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02$.

Let $b(x) = c(x)a(x) \bmod (x^4 + 1)$, then the modular multiplication with a fixed polynomial can be written as:

$$\begin{bmatrix} b_{0,0} \\ b_{0,1} \\ b_{0,2} \\ b_{0,3} \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} a_{0,0} \\ a_{0,1} \\ a_{0,2} \\ a_{0,3} \end{bmatrix} \tag{12.1}$$

For the decryption process, we compute Inverse MixColumns using the constant polynomial $w(x) = w_3x^3 + w_2x^2 + w_1x + w_0$, with coefficients $w_0(x) = x^3 + x^2 + x$, $w_1(x) = x^3 + 1$, $w^2(x) = x^3 + x^2 + 1$, $w_3(x) = x^3 + x + 1$, and reduced modulo $M(x) = x^4 + 1$, which is multiplied by each column of a block. The equivalent matrix representation is

$$\begin{bmatrix} b_{0,0} \\ b_{0,1} \\ b_{0,2} \\ b_{0,3} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} a_{0,0} \\ a_{0,1} \\ a_{0,2} \\ a_{0,3} \end{bmatrix} \tag{12.2}$$

12.3.4 Add Round Key (ARK) Step

The output of MC is XORed with the round key derived from the user key. The ARK step is symmetric for encryption and decryption. The only difference is that the sub-key rounds are applied in reverse order for decryption.[4] Each one of the above described transformations BS, SR, MC and ARK are invertible [28]. Let us call them IBS, ISR, IMC and IARK, respectively. Then the AES encryption/decryption procedures can be described as follows:

1. ARK, using the 0th round key.
2. Nine rounds of BS, SR, MC, ARK, using round keys 1–9.
3. A final round: BS, SR, ARK, using the 10th round key.

Similarly, the decryption is computed as follows:

1. ARK, using the 10th round key.
2. Nine rounds of IBS, ISR, IMC, IARK, using round keys 9–1.
3. A final round: IBS, ISR, ARK, using the 0th round key.

12.3.5 Key Scheduling Algorithm

The round keys are obtained through the expansion of secret user key by attaching recursively the 4-byte word $k_i = (k_{0,i}, k_{1,i}, k_{2,i}, k_{3,i})$ to the user key. The original user key consists of 128 bits arranged as a 4×4 matrix of bytes. Let $w[0], w[1], w[2]$ and $w[3]$ be the four columns of the original user key. Then, those four columns are recursively expanded to obtain 40 more columns as follows:

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if } i \bmod 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \text{otherwise} \end{cases} \tag{12.3}$$

where $T(w[i-1])$ is a non-linear transformation of $w[i-1]$ computed as follows. Let w, x, y and z be the elements of the column $w[i-1]$, then

1. Shift cyclically the elements to obtain x, y, z and w.
2. Replace each byte by a byte using the S-Box as $S(x), S(y), S(z), S(w)$.
3. Compute the round constant *rcon*, defined as $r(i) = 02^{(i-4)/4}$ over $GF(2^8)$.

[4] However, efficient implementations of AES encryptor/decryptor cores require to append the IMC step to the generation of round keys for decryption.

Then $T(w[i-1])$ is the column vector, $(S(x) \oplus r(i), S(y), S(z), S(w))$. In this way, columns from $w[4]$ to $w[43]$ are generated from the first four columns. Hence, the ith round key consists of the columns

$$(w(4i), w(4i+i), w(4i+2), w(4i+3)) \tag{12.4}$$

In [47] several optimizations based on redundant computation for parallelizing the Key Scheduling process were implemented. As a result, above four steps can be reduced to only two steps [47].

Step 1	Step 2	(12.5)
$k'_0 = k_0 \oplus SBox(k_{13}) \oplus rcon;$	$k'_4 = k_4 \oplus k'_0;$	
	$k'_8 = k_8 \oplus k_4 \oplus k'_0;$	
	$k'_{12} = k_{12} \oplus k_8 \oplus k_4 \oplus k'_0;$	
$k'_1 = k_1 \oplus SBox(k_{14});$	$k'_5 = k_5 \oplus k'_1;$	
	$k'_9 = k_9 \oplus k_5 \oplus k'_1;$	
	$k'_{13} = k_{13} \oplus k_9 \oplus k_5 \oplus k'_1;$	
$k'_2 = k_2 \oplus SBox(k_{15});$	$k'_6 = k_6 \oplus k'_2;$	
	$k'_{10} = k_{10} \oplus k_6 \oplus k'_2;$	
	$k'_{14} = k_{14} \oplus k_{10} \oplus k_6 \oplus k'_2;$	
$k'_3 = k_3 \oplus SBox(k_{12});$	$k'_7 = k_7 \oplus k'_3;$	
	$k'_{11} = k_{11} \oplus k_7 \oplus k'_3;$	
	$k'_{15} = k_{15} \oplus k_{11} \oplus k_7 \oplus k'_3;$	

12.4 A Background in Binary Extension Finite Fields

12.4.1 Rings

A ring R is a set whose objects can be added and multiplied, satisfying the following conditions:

- Under addition, R is an additive (Abelian) group.
- For all $x, y, z \in R$ we have,

$$\begin{aligned} x(y+z) &= xy+xz \\ (y+z)x &= yx+zx \end{aligned}$$

- For all $x, y \in R$, we have $(xy)z = x(yz)$.
- There exists an element $e \in R$ such that $ex = xe = x$ for all $x \in R$.

The integer numbers, the rational numbers, the real numbers and the complex numbers are all rings.

An element x of a ring is said to be invertible if x has a multiplicative inverse in R, that is, if there is a unique $u \in R$ such that $xu = ux = 1$. 1 is called the *unit element* of the ring.

12.4.2 Fields

A field is a ring in which the multiplication is commutative and every element except 0 has a multiplicative inverse. We can define the field F with respect to the addition and the multiplication if

- F is a commutative group with respect to the addition.
- $F \setminus \{0\}$ is a commutative group with respect to the multiplication.
- The distributive laws mentioned for rings hold.

12.4.3 Finite Fields

A finite field or *Galois field* denoted by $GF(q = p^m)$ is a field with characteristic p and a number q of elements. Such a finite field exists for every prime p and positive integer m and contains a subfield having p elements. This subfield is called *ground field* of the original field. For every non-zero element $\alpha \in GF(q)$, the identity $\alpha^{q-1} = 1$ holds. Furthermore, an element $\alpha \in GF(q^m)$ lies in $GF(q)$ itself if and only if $\alpha^q = \alpha$.

In the following we will only consider binary extension fields, where $q = 2^m$, also known as finite fields of characteristic two or simply binary fields.

12.4.4 Binary Finite Field Arithmetic

In the following we will use the *polynomial basis* representation of the binary finite fields elements. We represent each element as a binary string $(a_{m-1} \ldots a_2 a_1 a_0)$, which is equivalently considered a polynomial of degree less than n:

$$a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0. \tag{12.6}$$

Addition is by far the less-costly field operation, whereas multiplication is arguably the most important arithmetic operation.

The addition of two elements $a, b \in F$ is simply the addition of two polynomials, where the coefficients are added in $GF(2)$, or equivalently, using the bit-wise XOR operation on the vectors a and b. The multiplication of two field elements can be accomplished as follows.

Let $A(x), B(x)$ be elements of $GF(2^m)$ and let $P(x)$ be the degree m irreducible polynomial generating $GF(2^m)$. Then, the field product $C'(x) \in GF(2^m)$ can be obtained by first computing the polynomial product $C(x)$ as

$$C(x) = A(x)B(x) = \left(\sum_{i=0}^{m-1} a_i x^i\right)\left(\sum_{i=0}^{m-1} b_i x^i\right) \tag{12.7}$$

Followed by a reduction operation, performed in order to obtain the $(m-1)$-degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \bmod P(x) \tag{12.8}$$

Once the irreducible polynomial $P(x)$ is selected and fixed, the reduction step can be accomplished using only XOR gates.

A particular case of field multiplication is that of multiplying an arbitrary field element $A(x)$ by the field element x, an operation sometimes called *xtimes*, that is frequently used in block cipher modes of operations and forms part of the AES specification. The operation *xtimes* can be accomplished very efficiently by noticing that

$$xtimes(A) = x \cdot A(x) = x \cdot \sum_{i=0}^{m-1} a_i x^i = \sum_{i=0}^{m-1} a_i x^{i+1} \tag{12.9}$$

Therefore, if the most significant bit of A, namely a_{m-1}, is equal to zero, then $xtimes(A)$ can be accomplished by a single left shift of the original element A. On the other hand, if $a_{m-1} = 1$, we can simply add $P(x)$ to $C(x)$, thus reducing the power $a_{m-1}x^m$. As a concrete example consider the case when $P(x)$ is an irreducible pentanomial. Then, the *xtimes* operation can be computed as

$$xtimes(A) = \begin{cases} \sum_{i=0}^{m-2} a_i x^{i+1} & \text{if } a_{m-1} = 0 \\ \sum_{i=0}^{m-2} a_i x^{i+1} + (x^{k_2} + x^{k_1} + x^{k_0} + 1) & \text{if } a_{m-1} = 1 \end{cases} \tag{12.10}$$

The computational cost of the above equation is one left shift followed by possibly an XOR operation.

12.5 Traditional Modes of Operations

As it was already mentioned, a block cipher can only encrypt fixed length strings, but in real life, messages are of arbitrary lengths and are not restricted to the block length of a block cipher. A mode of operation is a specific way to use a block cipher for encrypting arbitrarily long messages. We now discuss some of the traditional modes of operations which have been in use for long. The five modes which we describe next are called the ECB (electronic code book), CBC (cipher block chaining), CFB (cipher feedback), OFB (output feedback) and CTR (counter). For ease of description we shall assume that the length of the plaintexts are multiples of the blocklength of the block cipher. We shall denote by n the blocklength and by m the number of blocks.

12.5.1 Electronic Code Book Mode

This is probably the simplest of all modes. In the electronic code book (ECB) mode the plaintext P is segmented as $P = P_1||P_2||\ldots||P_m$, where each P_i is an n-bit long block. Thereafter, the encryption function E_K is applied separately on each P_i. A schematic diagram of a ECB mode is shown in Figure 12.4.

12.5.2 Cipher Block Chaining Mode

In cipher block chaining (CBC) mode, the output of one block cipher is fed into the other block cipher along with the next block message. The algorithm below describes the mode and a pictorial description is provided in Figure 12.4b.

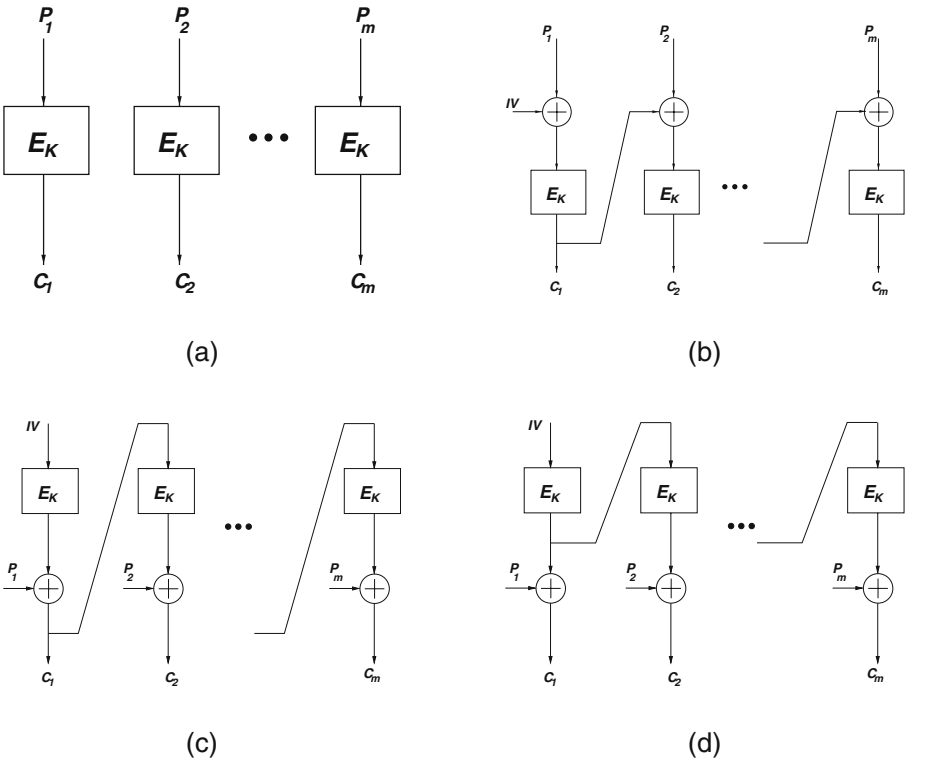

Fig. 12.4 The traditional modes of operation: (**a**) ECB, (**b**) CBC, (**c**) CFB, (d) OFB.

Algorithm CBC.Encrypt$_K^{\text{IV}}(P)$	**Algorithm** CBC.Decrypt$_K^{\text{IV}}(C)$
1. Partition P into $P_1, P_2, \ldots, P_m$	1. Partition C into $C_1, C_2, \ldots, C_m$
2. $C_1 \leftarrow E_K(P_1 \oplus \text{IV})$;	2. $P_1 \leftarrow E_K^{-1}(C_1) \oplus \text{IV}$
3. **for** $i \leftarrow 2$ to m	3. **for** $i \leftarrow 2$ to m
4. $\quad C_i \leftarrow E_K(P_i \oplus C_{i-1})$	4. $\quad P_i \leftarrow E_K^{-1}(C_i) \oplus C_{i-1}$
5. **end for**	5. **end for**
6. **return** $C_1, C_2, \ldots, C_m$	6. **return** $P_1, P_2, \ldots, P_m$

CBC takes as input m message blocks and an initialization vector (IV). During encryption, the output of the ith block depends on the previous $i-1$ blocks. So, CBC encryption is inherently sequential. The output of each block depends on all the previous blocks and thus provides more security than ECB. The sequential design does not allow a fully pipelined implementation for this mode. Note that CBC decryption is not sequential.

12.5.3 Cipher Feedback Mode

The encryption and decryption procedures for the cipher feedback mode (CFB) are described below. A pictorial description of the mode is provided in Figure 12.4c.

Algorithm CFB.Encrypt$_K^{\text{IV}}(P)$	**Algorithm** CFB.Decrypt$_K^{\text{IV}}(C)$
1. Partition P into $P_1, P_2, \ldots, P_m$	1. Partition C into $C_1, C_2, \ldots, C_m$
2. $C_1 \leftarrow E_K(\text{IV}) \oplus P_1$;	2. $P_1 \leftarrow E_K(IV) \oplus C_1$
3. **for** $i \leftarrow 2$ to m	3. **for** $i \leftarrow 2$ to m
4. $\quad C_i \leftarrow E_K(C_{i-1}) \oplus P_i$	4. $\quad P_i \leftarrow E_K(C_{i-1}) \oplus C_i$
5. **end for**	5. **end for**
6. **return** $C_1, C_2, \ldots, C_m$	6. **return** $P_1, P_2, \ldots, P_m$

In CFB mode also the cipher blocks are chained but the output is produced in a manner much different from that of CBC. For each block, the cipher produced is just XORed with the message. Due to such a kind of encryption, the encryption and decryption operations are similar. In case of decryption also the inverse block cipher calls are not required. Like CBC, OFB encryption and decryption are also inherently sequential. But an advantage in terms of implementation is that the block cipher decryption operation is not needed.

12.5.4 Output Feedback Mode

In output feedback mode (Figure 12.4d) unlike CFB, the output of the block cipher is fed back into the next block cipher. The algorithm is as shown below.

Algorithm CFB.Encrypt$_K^{\text{IV}}(P)$	**Algorithm** CFB.Decrypt$_K^{\text{IV}}(C)$
1. Partition P into $P_1, P_2, \ldots, P_m$	1. Partition C into $C_1, C_2, \ldots, C_m$
2. $X \leftarrow \text{IV}$;	2. $X \leftarrow \text{IV}$
3. **for** $i \leftarrow 1$ to m	3. **for** $i \leftarrow 1$ to m
4. $\quad X \leftarrow E_K(X)$;	4. $\quad X \leftarrow E_K(X)$;
5. $\quad C_i \leftarrow X \oplus P_i$	5. $\quad P_i \leftarrow X \oplus C_i$
6. **end for**	6. **end for**
7. **return** $C_1, C_2, \ldots, C_m$	7. **return** $P_1, P_2, \ldots, P_m$

In this mode the *IV* is repeatedly encrypted to get a stream of random bytes. Unlike the other modes described before in OFB no part of the plaintext is ever given as an input to the blockcipher. This makes this mode very similar to a stream cipher, where a stream cipher produces a stream of random bytes and these random strings are XORed with the plaintext to generate the cipher. The specific way in which the *IV* is encrypted in the mode also makes the algorithm sequential. Hence, as in the case of CFB, for both encryption and decryption operations, only a forward call of the block cipher (i.e., its encryption algorithm) is required.

12.5.5 Counter Mode

The counter (CTR) mode is a bit different from the other modes defined above. It takes in an *IV*, and in each iteration the value of the *IV* incremented by one gets encrypted. The ciphertext is produced by XORing the encryption results with the plaintext blocks.

Algorithm CTR.Encrypt$_K^{\text{IV}}(P)$	**Algorithm** CFB.Decrypt$_K^{\text{IV}}(C)$
1. Partition P into $P_1, P_2, \ldots, P_m$	1. Partition C into $C_1, C_2, \ldots, C_m$
2. $C_1 \leftarrow E_K(\text{IV}) \oplus P_1$;	2. $P_1 \leftarrow E_K(IV) \oplus C_1$
3. **for** $i \leftarrow 2$ to m	3. **for** $i \leftarrow 2$ to m
4. $\quad C_i \leftarrow E_K((IV+i) \bmod 2^n) \oplus P_i$	4. $\quad P_i \leftarrow E_K((IV+i) \bmod 2^n) \oplus C_i$
5. **end for**	5. **end for**
6. **return** $C_1, C_2, \ldots, C_m$	6. **return** $P_1, P_2, \ldots, P_m$

In terms of efficiency the CTR mode is better than CBC, OFB or CFB as in CTR the block cipher calls can be done in parallel. No feedback takes place in case of CTR so the input to the *i*th block cipher in no way depends on the output of the previous block ciphers. Also in CTR only the encryption algorithm is needed. Additionally, for performing the decryption operation, the inverse call of the block cipher is not needed.

12.6 Security Requirements for Modes of Operations

In Section 12.5 we discussed some modes which can be used to encrypt messages longer than the block length of the block cipher. Now we shall analyze some of these modes to see whether they are secure. Also we shall try to intuitively formulate security requirements for three important classes of modes.

Let us begin by analyzing the ECB mode. The ECB mode is not suitable for encrypting bulk messages as it can reveal much information about a message. We already illustrated this with the help of Figure 12.2a and its ECB encryption in 12.2b. From Figure 12.2b we see that the encryption reveals much information regarding the image. This is because in ECB, every block is encrypted using the same key and so all equal plaintext blocks gets encrypted into equal ciphertext blocks. So, if we encrypt a four-block message say P_1, P_2, P_3, P_4 where $P_1 = P_2$, then in the ciphertext blocks C_1, C_2, C_3, C_4 also C_1 would be equal to C_2. This is not desirable as the structure of the plaintext blocks gets revealed in the ciphertext blocks. In this particular example, an adversary can readily find out that the first two plaintext blocks were equal just by looking at the ciphertext. This limitation of ECB makes the encryption in Figure 12.2b look so similar with the figure itself. Thus, ECB is insecure.

The important question we would like to address now is when a mode can be considered secure. In order to answer that, we first establish a formal model of the adversary who tries to break the security of a mode.

12.6.1 The Adversary

To define security we need to formalize the goals and resources of an adversary. An adversary can have various goals, the strongest among them, being able to recover the keys that the encryption scheme uses. With knowledge of the key she can always decrypt all encrypted messages that goes through the public channel and can also replace encrypted messages with the encryption of messages of her choice. But the goal of key recovery is a very strong goal and without recovering the key also an adversary can predict some properties of the plaintext. Thus, an intuitive goal of a crypto-system should be that it leaks no information regarding the plaintext through the ciphertext.[5] On the other hand, the weakest goal that an adversary can have is being able to distinguish the ciphertext from random strings. Thus, if a crypto-system is strong enough that an adversary cannot accomplish this weak goal, then to an adversary the ciphertext would not in any way leak any information regarding the plaintext.

An adversary who wants to break the security of a symmetric crypto-system must be given access to some of the inputs or outputs of the system. The type of

[5] Note that we already showed that the electronic code book mode (ECB) leaks some important information regarding the plantext, which is undesirable.

information access the adversary has defines the power of the adversary. The adversary always has access to the ciphertexts as he can eavesdrop the public channel and know the ciphertext. If an adversary only has access to the ciphertext we call the attack mounted by the adversary as a *ciphertext-only attack*. Additionally the adversary may know which messages produce these ciphertexts, such an attack is a *known plaintext attack*. If the adversary can choose the plaintexts whose encryptions he wants, then the attack is a *chosen plaintext attack*. There can be adversaries who can choose the ciphertexts and get the corresponding plaintexts for those ciphertexts. Such a kind of attack is called a *chosen ciphertext attack*. The strongest adversary is the one who can adaptively choose messages (ciphertext) and get their encryptions (decryptions). Such adversaries are called *adaptive chosen plaintext* (respectively ciphertext) adversaries.

To formalize things we shall view the adversary as a polynomial time probabilistic algorithm with certain resources. An adaptive chosen plaintext adversary should be supplied with ciphertexts corresponding to the plaintexts of her choice. To do this we allow the adversary to communicate with the encryption algorithm, i.e., she is given access to the encryption scheme as a black box where she gives some inputs and obtains the corresponding outputs but has no access to the internal workings of the scheme. We call such an access as an oracle access. An adversary may be given access to one or more oracles, as in case of an adaptive chosen plaintext and chosen ciphertext adversary the adversary should be given access to both the encryption and decryption oracles so that he can adaptively obtain encryptions and decryptions of his choice.

12.6.2 Privacy Only Modes

With the above characterization of the adversary we now try to define the security requirements for a class of modes called *privacy only modes*. In a privacy only mode the goal is to create ciphertexts such that the adversary by knowing the ciphertexts can have no knowledge of the plaintext. So, such an adversary has access to the ciphertexts and also we assume that the adversary can have access to the ciphertexts corresponding to plaintexts of her choice. To model security of a privacy only mode we will give the adversary access to two oracles. The first one is the encryption algorithm (i.e., the mode) and the second one is an algorithm which when given an input of a plaintext of length m returns m random bits. Thus, the adversary has two oracles, one of which is the real mode and the other returns random strings. The adversary can query these oracles without repeating any query and his task is to distinguish between these two oracles. If the probability with which any efficient adversary can distinguish between these two oracles is small then the mode can be considered secure in terms of privacy.

With this definition of privacy let us try to analyze the security of the CBC mode of operation shown in Figure 12.4b. Let us recall that in the CBC mode, the algorithm takes as input a key, an IV and the plaintext. The key is secret but in general

we may assume the IV is a public quantity. If we allow an adversary A to freely choose the IVs along with the plaintexts then CBC is also not secure in terms of privacy. An easy distinguishing attack can be mounted by A in the following manner. A provides two encryption queries and gets the corresponding ciphertexts as below:

$$\text{Query 1: } \mathrm{IV}^1; P_1^1, P_2^1 \rightarrow C_1^1, C_2^1$$
$$\text{Query 2: } \mathrm{IV}^2; P_1^2, P_2^2 \rightarrow C_1^2, C_2^2$$

Here P_i^j and C_i^j represent a block (say n bits if the block length of the block cipher is n) of plaintext and ciphertext, respectively. Additionally we assume the following restrictions on the queries:

$$P_1^1 = P_1^2 = \mathrm{IV}^2 = \mathrm{IV}^1$$
$$P_2^2 = C_1^1$$

For such a set of queries C_1^2 will always be equal to C_2^2. This happens because for the first query C_1^1 will be the block cipher output for all zero string and for the second query both C_1^2 and C_2^2 will be the encrypted output of all zero strings. Thus we see that if A is freely allowed to choose IVs then he can easily distinguish a CBC output from random strings, so CBC in this setting is not secure in terms of privacy.

To make CBC secure, we need to put a restriction on the usage of IV. First, a key and IV pair is never to be repeated. Moreover, if in CBC encryption one replaces the IV by the encryption of the *IV* (i.e., $E_K(\mathrm{IV})$) then CBC is secure in terms of privacy. Note that the IV is still a public quantity and we may allow an adversary to obtain encryptions of messages with IVs of his choice, but he is not allowed to obtain encryption of two different messages using the same IV. The IV used in this manner is called a *nonce*.

The formalization of the security requirement of privacy only modes along with the analysis of the security of the traditional modes of operation were first provided in [4] and the security of CBC with the IV as a nonce was proved in [49].

12.6.3 Authenticated Encryption

The security provided by privacy only modes may not be enough in certain scenarios. Recall, for defining privacy we assumed the adversary to be an adaptive chosen plaintext adversary whose task was to distinguish the output of the mode from random strings. Thus, if an adversary sees only ciphertexts from a secure privacy only mode, he cannot determine anything meaningful from the ciphertexts. But, if we assume that the adversary wants to tamper the ciphertexts which goes through the public channel he can always do so. In a privacy only mode the receiver has no way to determine whether she received the ciphertext that was originally sent by the sender. This forms a major limitation of privacy only modes.

To overcome this limitation we need to add some other functionality to a mode so that the receiver of a message can verify whether she had obtained the ciphertext sent by the sender. This is obtained by a tag. A tag can be considered as a checksum of the message that was used to generate the ciphertext. A sender after decrypting the ciphertext can always compute the tag and match the tag which she computed using the decrypted message with the tag that she received. If the tags do not match the receiver can know that a tampering of the ciphertext has taken place during the transit. This functionality in the symmetric setting is called authentication and the modes which provide both privacy and authentication are called authenticated encryption modes.

Thus an authenticated encryption mode can be seen as a pair of algorithms $(\boldsymbol{E},\boldsymbol{D})$ where $\boldsymbol{E}$ produces the ciphertext $\mathsf{C} = (C, \mathsf{tag})$ when given a plaintext P as an input. While the decryption algorithm $\boldsymbol{D}$ on an input C produces the corresponding plaintext P or outputs INVALID if the computed tag does not match tag.

The security requirement of AE schemes are a bit different from privacy only modes. For an AE scheme we do not want an adversary to be able to distinguish ciphertexts produced from plaintexts chosen by her adaptively. So the security requirement for privacy only modes is also a requirement for AE schemes, but additionally we want that the adversary should not be able to construct any ciphertext which will get decrypted. To model this requirement we assume that the adversary is given a number of ciphertext, tag pairs for plaintexts of his/her choice and after observing these ciphertexts the task of the adversary is to forge, i.e., to construct a ciphertext tag pair which on input to the decryption algorithm does not produce INVALID. An AE mode is considered secure in the sense of authenticity if the probability of forging of any efficient adversary is low. A secure AE scheme is needed to be secure both in the sense of privacy and authenticity.

Another class of AE schemes are called authenticated encryption with associated data (AEAD). These schemes can be useful in certain realistic scenarios. Like if we consider network packets, we do not want to encrypt the headers but we want to authenticate the headers so that they cannot be tampered. Such schemes take as input the message and an associated data (the packet header in this case), the message is only encrypted but the tag is produced both with the message and the header. Most AE schemes can be easily converted into AEADs.

12.6.4 Disk Encryption Schemes

Now, let us look at another application. Suppose we want to encrypt all data present in the hard disk of a computer. Whenever there is a disk read then the particular disk sector is decrypted and returned, similarly whenever some given data need to be written to the disk the specific sector is encrypted and then written. The encryption and decryption operations get done by the disk controller, which is a low-level device having no knowledge of the files, directories, etc. maintained by the operating system. Each sector is considered as a message.

In this setting a very important limitation of the encryption algorithm to be used is that the encryption should be length preserving, i.e., the length of the ciphertext and plaintext should be equal. This limitation dictates that AE schemes cannot be used for such applications, as in AE schemes always there is a ciphertext expansion. A privacy only kind of mode is generally length preserving, but the security it provides will not be enough for disk encryption schemes, as we do not want an adversary to tamper the data present in the disk without our knowledge. So, we need schemes in which the ciphertext produced will be indistinguishable from random strings to any adversary who can access ciphertexts corresponding to the plaintexts of his/her choice. Additionally if an adversary chooses ciphertexts and gets plaintexts corresponding to those ciphertexts, she should be unable to distinguish those plaintexts from random strings.

It should be noticed that this adversary is different from the adversary we discussed in case of privacy only and AE modes. In privacy only and the AE schemes the adversary had the freedom to choose plaintexts and get the corresponding ciphertexts and her task was to distinguish the ciphertexts from random strings. Here we are giving the adversary freedom to choose ciphertexts also. This adversary is an adaptive chosen plaintext and chosen ciphertext adversary. So, in this scenario, if the adversary plans to change the original ciphertext in the disk with some ciphertexts of her choice, then the decrypted plaintexts will be indistinguishable from random. Thus, the adversary cannot create any ciphertext which will get decrypted into something meaningful, in other words whatever ciphertext she creates will look like random when it gets decrypted.

To define security of disk encryption (DE) schemes we assumed a more powerful adversary than in case of AE schemes, but the security provided by DE schemes is less than that of AE schemes. Recall, in AE schemes the adversary has two tasks, one of distinguishing and another of forging. If an AE scheme is secure against forgery attacks then the probability with which an adversary can create a valid ciphertext (i.e., the probability with which she can tamper a ciphertext which still gets decrypted) is very low. But in case of DE schemes the adversary can tamper the ciphertext and there is no mechanism in the scheme which can detect the tampering. But the security definition guarantees that if such a tampering takes place then the corresponding plaintext will be random. So, a high-level application which uses the plaintext can detect the tampering. As most applications assume certain structure on the data it uses and a tampering of the data will violate the structure, the probability that an efficient adversary creates a ciphertext which will retain the structure in the plaintext is low.

As we discussed in DE schemes the data in a sector are considered as a plaintext/ciphertext. Thus if two sectors contain the same data they would get encrypted into the same ciphertexts and the adversary can readily get the information that the two sectors contain same data. We would not like the adversary to know such information. To assure that this does not happen the DE schemes take in a quantity called tweak along with the plaintext. The tweak may be considered as a type of associated data. The tweak is not secret and there is no restriction about repetition of the tweak as in nonces. The encryption of a message depends on the tweak used to encrypt it.

In DE schemes the sector address is considered as the tweak. So, if two different sectors contain the same message also they would have different sector addresses and thus different tweaks, so their encryptions would be different.

12.6.5 Security Proofs

We have intuitively discussed some of the security requirements of modes of operations. Note that we have not provided with formal definitions of security which can be done. The basic building block of a mode of operation is a block cipher. So the security of a mode of operation heavily depends on the security of the underlying block cipher. It is a pity that the security of primitives like block ciphers cannot be formally proved. Instead, for block ciphers we just assume security based on the facts that they can resist all known attacks.

A mode of operation is built using a block cipher and the security of a mode is thus reduced in a suitable way to the security of the block cipher. Thus, assuming the block cipher to be secure in a certain way, the security of the mode is derived using the (presumed) security of the block cipher. Such a reduction is called a security proof. We shall not discuss security proofs in this chapter, but any modern mode has a security proof associated with it and it proves an upper bound on advantage of any efficient adversary in breaking the security of that mode [10, 45].

12.7 Some Modern Modes

We already discussed security requirements of three important kinds of modes. Of these three modes, the privacy only modes are of limited interest as they do not provide security against active adversaries who can tamper the ciphertexts. The AE modes and DE modes are of much interest in the current days. There are many modes proposed till date. In Table 12.1 we list some secure AE and DE modes.

AE modes can be classified according to the number of passes over the data it requires. Easiest way to obtain an AE mode is to use two algorithms, one for computing the tag and the other for encrypting. If these two algorithms are used separately they obviously need two passes over the message and additionally two keys will be required. This paradigm is called generic composition and was first formally analyzed in [5].

The most efficient AE modes are the one pass AE modes. As the name suggests they use only one pass over the data. Some single pass AE schemes proposed till date are IACBC [29], IAPM [30], OCB [50], XCBC and XECB [20]. Also a generalization of the OCB construction was provided in [10]. Out of these modes OCB is probably the most efficient and optimized AE mode. We provide a description of OCB in Section 12.7.1

Table 12.1 Some secure modes: AE stands for authenticated encryption and DE for disk encryption.

Mode	Source	Type	Notes
OCB	[50]	AE	One Pass
IAPM	[30]	AE	One pass
IACBC	[29]	AE	One Pass
XCBC	[20]	AE	One Pass
XECB	[20]	AE	One Pass
CCM	[43]	AE	Two Pass
EAX	[6]	AE	Two Pass
CWC	[31]	AE	Two Pass
GCM	[41]	AE	Two Pass
CMC	[24]	DE	Encrypt-Mask-Encrypt
EME	[25]	DE	Encrypt-Mask-Encrypt
EME*	[22]	DE	Encrypt-Mask-Encrypt
PEP	[11]	DE	Hash-ECB-Hash
HCTR	[54]	DE	Hash-CTR-Hash
HCH	[12]	DE	Hash-CTR-Hash
TET	[23]	DE	Hash-ECB-Hash
HEH	[52]	DE	Hash-ECB-Hash

Other than the one pass schemes there exist other AE schemes which require two passes over the data. For such modes, in one pass the ciphertext is computed and in the other pass the tag is computed. Surely such modes are inefficient than the one pass modes. All known one pass schemes except [10] are covered by patent claims.[6] That is why two pass schemes are still of interest though one pass schemes exist. Some of the two pass AE modes are CCM [15, 43], EAX [6], GCM [41] etc. We shall discuss a two pass AE mode called CCM mode in detail including its hardware implementation in Section 12.8.

Till date there are 10 disk encryption modes proposed. They are CMC [24], EME [25] EME* [9, 22], XCB [39], ABL [40], HCTR [54], PEP [11], HCH [12], TET [23] and HEH [52]. The construction of these modes falls under three basic paradigms. The first paradigm is called encrypt-mask-encrypt where two layers of encryption are used with a layer of masking in between. CMC, EME and EME* fall under this category. Another way of construction is to use electronic code book-type encryption in between two hash layers, such constructions are called as hash-ECB-hash constructions. PEP and TET fall under this category of constructions. The other category, hash-counter-hash, uses a counter mode in between two hash layers. HCTR, XCB, ABL and HCH fall under this category of constructions. We provide a description of one DE mode EME in Section 12.7.2 which falls under the encrypt-mask-encrypt category.

[6] There is no known patent granted or pending on [10], but it may be covered by some existing patent claims unknown to the authors.

12.7.1 The Offset Codebook Mode

The offset codebook (OCB) mode was proposed by Rogaway and Black [50]. This is a fully defined efficient mode which provides both privacy and authenticity. The original OCB mode was modified a bit in [48] and called OCB1. OCB1 is not much different from the original OCB. But certain tricks in the construction help to reduce the complexity of the security proof. Also the description of OCB1 is easier than the original OCB proposal.

Figure 12.5 shows the encryption and decryption algorithm using OCB1. The encryption algorithm takes in a m block message (the last block can be an incomplete block, i.e., the last block can have a block length less than the block length of the block cipher), a block cipher key K and a nonce N. It produces a m block ciphertext along with a τ bit tag. If n is the block length of the block cipher E_K, then all n-bit strings in the algorithm are viewed as elements in $GF(2^n)$. So, all n-bit strings in the algorithm can be seen as polynomials of degree less than n whose coefficients are from $GF(2)$ (see Section 12.4 for a detailed discussion). The operation $\oplus$ is addition in the field $GF(2^n)$ and the operations $xE_K(N)$ and $(x+1)E_K$ are multiplications of the polynomials x and $1+x$ with the polynomial $E_K(N)$ modulo a fixed irreducible polynomial in $GF(2^n)$. The algorithm is self-explanatory, but certain points are important to see. The encryption of the last block (i.e., the mth block in this case) is different from the encryption of the other blocks. In step 10 of the algorithm, C_m would be t bits long if P_m is t bit long. The operation $C_m 0^*$ in step 11 means to add $(n-t)$ zeros to C_m to make C_m a full block. These discussions are all valid for the decryption algorithm also.

Algorithm OCB1.Encrypt$_K^N(P)$	**Algorithm** OCB1.Decrypt$_K^N(\mathscr{C})$
1. Partition P into $P_1, P_2, \ldots, P_m$	1. Partition $\mathscr{C}$ into $C_1, C_2, \ldots, C_m, T$
2. $\Delta \leftarrow xE_K(N)$	2. $\Delta \leftarrow xE_K(N)$
3. $\Sigma \leftarrow 0^n$	3. $\Sigma \leftarrow 0^n$
4. **for** $i = 1$ to $m-1$,	4. **for** $i = 1$ to $m-1$,
5. $\quad C_i \leftarrow E_K(P_i \oplus \Delta) \oplus \Delta$	5. $\quad P_i \leftarrow E_K^{-1}(C_i \oplus \Delta) \oplus \Delta$
6. $\quad \Delta \leftarrow x\Delta$	6. $\quad \Delta \leftarrow x\Delta$
7. $\quad \Sigma \leftarrow \Sigma \oplus P_i$	7. $\quad \Sigma \leftarrow \Sigma \oplus P_i$
8. **end for**	8. **end for**
9. Pad $\leftarrow E_K(\mathsf{len}(P_m) \oplus \Delta)$	9. Pad $\leftarrow E_K(\mathsf{len}(C_m) \oplus \Delta)$
10. $C_m \leftarrow P_m \oplus$ Pad	10. $P_m \leftarrow C_m \oplus$ Pad
11. $\Sigma \leftarrow \Sigma \oplus C_m 0^* \oplus$ Pad	11. $\Sigma \leftarrow \Sigma \oplus C_m 0^* \oplus$ Pad
12. $\Delta \leftarrow (1+x)\Delta$	12. $\Delta \leftarrow (1+x)\Delta$
13. Tag $\leftarrow E_K(\Sigma \oplus \Delta)$	13. Tag $\leftarrow E_K(\Sigma \oplus \Delta)$
14. $T \leftarrow$ Tag[first τ bits]	14. $T' \leftarrow$ Tag[first τ bits]
15. **return** $\mathscr{C} \leftarrow C_1 \| C_2 \| \ldots \| C_m \| T$	15. **if** $T = T'$ **return** $P \leftarrow P_1 \| P_2 \| \ldots \| P_m$ **else return** INVALID

Fig. 12.5 Encryption and decryption using OCB1.

OCB1 requires $m+1$ block cipher calls to encrypt a m block message. The other operations it requires have insignificant computational overhead. It requires only one pass over the data and can produce cipher in an online manner. Note that OCB1 requires only the length information of the last block to encrypt or decrypt. Also OCB1 uses only one block cipher key. Assuming a nonce respecting adversary (i.e., an adversary who does not repeat nonces) OCB can be proved to be secure in terms of both privacy and authenticity. These discussions are also valid for the original OCB.

12.7.1.1 Hardware Implementation Aspects of OCB

The parallel nature of the OCB mode of operation allows us to use a pipeline approach when implementing it in hardware. Furthermore, as is discussed in Section 12.4, the operation $xE_K(N)$ of algorithm of Figure 12.5 can be implemented at a negligible computational cost in hardware. In the following, we give a rough estimation of the hardware implementation cost of the OCB mode of operation.

Let us assume that we have an AES block cipher encryption core that uses a pipeline architecture of 10 stages. Then, referring to the algorithm of Figure 12.5, step 2 must be computed in a sequential fashion, implying that 10 clock cycles will be required for calculating Δ. Thereafter, the $m-1$ block cipher calls in steps 4–9 can be accomplished using the benefits of the parallelism associated to the pipeline architecture. So, we can argue that all the C_i for $i = 1, 2, \ldots, m$, can be computed in about $(m-1)+10$ clock cycles. Finally the tag computation of step 13 will require 10 extra clock cycles. Hence, according to the above analysis, we could achieve both authentication and encryption after about $(m-1)+20$ clock cycles when using the OCB mode of operation.

12.7.2 ECB-Mask-ECB Mode

Now we will discuss a disk encryption mode called ECB-mask-ECB (EME) [25]. As the name suggest, the mode consists of two electronic code book layers with a masking layer in between. The encryption and decryption algorithm are given in Figure 12.7. A pictorial description of EME is given in Figure 12.6.

EME takes in a m block message along with a tweak T. Note that the tweak here is different from the nonce N in case of OCB. There is no restriction regarding repetition of the tweak. Here also if the block length of the block cipher used is n then each n-bit string in the algorithm is considered as an element in the field $GF(2^n)$, i.e., they can be treated as polynomials of degree less than n with coefficients from the field $GF(2)$ and the operations x^iL and x^iM denote the multiplication of the polynomial x^i with the polynomials L and M, respectively, modulo a fixed irreducible polynomial.

The algorithm is self-explanatory, but an important feature to note is that in the masking layer, the mask M is dependent on all the plaintext blocks and the mask is

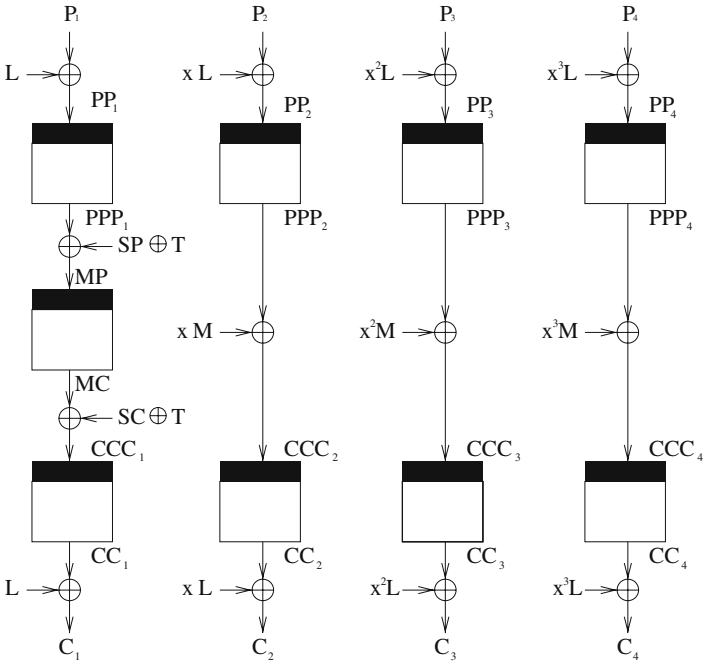

Fig. 12.6 Encryption of four blocks of plaintext using EME. Here, $L = xE_K(0^n)$, $SP = PPP_2 \oplus PPP_3 \oplus PPP_4$, $M = MP \oplus MC$ and $SC = CCC_2 \oplus CCC_3 \oplus CCC_4$.

distributed to all the blocks suitably. This makes each block of ciphertext dependent on all blocks of plaintexts. This is a necessary mechanism for any disk encryption mode.

EME has some message length restrictions. If the block length of the underlying block cipher is n then EME cannot encrypt more than n blocks of messages. Also the message length should always be a multiple of n. This message length restriction may not amount to a serious restriction in case of disk encryption scenarios as generally sector lengths are 512 bytes long.

To encrypt a m block message EME requires $2m+1$ block cipher calls, the other computational overheads are not significant. EME like all other DE modes needs to process the whole plaintext before it can output any ciphertext. Thus it is not an *online* mode of operation. EME is proved to be a secure tweakable enciphering scheme.

12.7.2.1 Hardware Implementation Aspects of EME

The EME mode of operation can be partially implemented in parallel in a hardware implementation. However, we stress that some computations of the algorithm of Figure 12.7 represent a bottleneck from the hardware implementation perspective as is discussed next.

Algorithm EME.Encrypt$_K^T(P)$	**Algorithm** EME.Decrypt$_K^T(C)$
1. Partition P into $P_1, P_2, \ldots, P_m$	1. Partition C into $C_1, C_2, \ldots, C_m$
2. $L \leftarrow xE_K(0^n)$	2. $L \leftarrow xE_K(0^n)$
3. **for** $i \leftarrow 1$ to m **do**	3. **for** $i \leftarrow 1$ to m **do**
4. $PP_i \leftarrow x^{i-1}L \oplus P_i$	4. $CC_i \leftarrow x^{i-1}L \oplus C_i$
5. $PPP_i \leftarrow E_K(PP_i)$	5. $CCC_i \leftarrow E_K^{-1}(CC_i)$
6. **end for**	6. **end for**
7. $SP \leftarrow PPP_2 \oplus PPP_3 \oplus \ldots PPP_m$	7. $SC \leftarrow CCC_2 \oplus CCC_3 \oplus \ldots CCC_m$
8. $MP \leftarrow PPP_1 \oplus SP \oplus T$	8. $MC \leftarrow CCC_1 \oplus SC \oplus T$
9. $MC \leftarrow E_K(MP)$	9. $MP \leftarrow E_K^{-1}(MC)$
10. $M \leftarrow MP \oplus MC$	10. $M \leftarrow MP \oplus MC$
11. **for** $i \leftarrow 2$ to m **do**	11. **for** $i \leftarrow 2$ to m **do**
12. $CCC_i \leftarrow PPP_i \oplus x^{i-1}M$	12. $PPP_i \leftarrow CCC_i \oplus x^{i-1}M$
13. **end for**	13. **end for**
14. $SC \leftarrow CCC_2 \oplus CCC_3 \oplus \ldots CCC_m$	14. $SP \leftarrow PPP_2 \oplus PPP_3 \oplus \ldots PPP_m$
15. $CCC_1 \leftarrow MC \oplus SC \oplus T$	15. $PPP_1 \leftarrow MP \oplus SP \oplus T$
16. **for** $i \leftarrow 1$ to m **do**	16. **for** $i \leftarrow 1$ to m **do**
17. $CC_i \leftarrow E_K(CCC_i)$	17. $PP_i \leftarrow E_K(PPP_i)$
18. $C_i \leftarrow x^{i-1}L \oplus CC_i$	18. $P_i \leftarrow x^{i-1}L \oplus PP_i$
19. **end for**	19. **end for**
20. **return** $C_1, C_2, \ldots, C_m$	20. **return** $P_1, P_2, \ldots, P_m$

Fig. 12.7 Encryption and decryption using EME.

As we did in the analysis of OCB, let us assume that we have an AES block cipher encryption core that uses a pipeline architecture of 10 stages. Then, referring to the algorithm of Figure 12.7, the computation of the parameter L in step 2 must be accomplished in a sequential fashion, implying that at least 10 clock cycles will be required for completing that calculation. Thereafter, the m block cipher calls included in steps 3–6 can be accomplished using the benefits of the parallelism associated to the pipeline approach. So, we can argue that all the PPP_i for $i = 1, 2, \ldots, m$ can be computed in about $(m-1)+10$ clock cycles. On the contrary, the cipher call in step 9 for obtaining MC must be performed in a sequential fashion, which implies 10 extra clock cycles. Similarly, the $m-1$ block cipher calls in steps 11–13 represent a computational effort of about $(m-2)+10$ clock cycles, whereas the last block cipher call in step 17 implies 10 clock cycles more.

In summary, the computational cost of the algorithm in Figure 12.7 can be estimated in about $2m-3+50$ clock cycles. Considering that for a typical EME application, the plaintext will have a length of 32 blocks,[7] then the EME algorithm of Figure 12.7 will encrypt a disk sector in about 111 clock cycles when using a pipeline AES hardware architecture. Some precomputations may save some cost in EME. As L is a quantity only dependent on the key K, L can be easily precomputed thus saving some clock cycles. A more detailed description of the hardware design of EME along with designs of other DE schemes can be found in [37, 38].

[7] Here we are assuming that the size of a disk sector is 512 bytes or thirty-two 128-bit AES blocks.

12.8 The CCM Mode: A Case Study

Here we shall discuss a mode called CCM in detail including its hardware implementation. We shall design CCM with AES as the underlying block cipher. For a summary of the AES algorithm specification we refer the interested reader to Section 12.3.

The rest of this section is organized as follows. In Section 12.8.1 we briefly describe the CCM mode of operation. Then, in Section 12.8.2, we present a reconfigurable hardware implementation of an AES sequential encryptor core. In Section 12.8.3, we give a design description of the AES–CCM mode reconfigurable hardware implementation reported in [34]. Finally, in Section 12.8.4 we compare the design described in this chapter with other architectures already reported in the open literature.

12.8.1 The CCM Mode

CCM stands for counter with CBC–MAC. This means that two different modes are combined into one, namely, the CTR mode and the CBC–MAC. CCM is a generic authenticated encrypt block cipher scheme. It has been specifically designed for being used in combination with a 128-bit block cipher, such as AES. CCM mode can be easily extended to other block sizes, but this would require further definitions not to be addressed here.

CCM mode was proposed by Whiting et al. [15]. Their original paper was sent to NIST for evaluation as a generic new mode. Presently, it has become part of the new 802.11i IEEE standard [43]. CCM is an authenticated encryption scheme which also supports associated data.

The generic CCM mode allows user definition of two main parameters. The first choice is M, the size of the tag or the authentication field. Selecting an adequate value for M involves a trade-off between message expansion and the probability that an attacker can undetectably modify the message. Valid values for M are 4, 6, 8, 10, 12, 14 and 16 bytes. This parameter is encoded as $(M-2)/2$.

In the rest of this section we will use $|P|$ to indicate the length in bytes of the plaintext message P. The parameter L gives the size in bytes of the field that indicates the numerical value of $|P|$. This value involves a trade-off between the maximum message size and the size of the *nonce*, which is an unique integer value associated with each message. The value of L ranges from two to eight.

12.8.1.1 CCM Input Parameters

Before sending a message, a sender must provide the following information:

- A suitable encryption key K for the block cipher to be used.
- A nonce N of $15-L$ bytes. Nonce value must be unique, meaning that the set of nonce values used with any given key shall not contain duplicate values.

- The message P consisting of a string of $|P|$ bytes where $0 \leq |P| < 2^{8L}$.
- Additional authenticated data a, consisting of a string of $|a|$ bytes where $0 \leq |a| < 2^{64}$. This additional data are authenticated but not encrypted and are not included in the output of this mode.

12.8.1.2 CCM Authentication

The first step consists of computing the tag or the authentication field T. This is done using CBC–MAC mode [15, 43]. We first define a sequence of blocks $B_0, B_1, \ldots, B_m$, and thereafter CBC–MAC is applied to those blocks so that the authentication field T can be obtained.

The first block B_0 is formatted as shown in Figure 12.8a, where $l(P)$ is encoded in most-significant-byte first order.

Within the first block B_0, the *Flags* field is formatted as shown in Figure 12.8b The *Reserved bit* field is reserved for future expansions and should always be set to zero. The *Adata* bit is set to zero if $l(a) = 0$ and set to one if $l(a) > 0$.

Authentication data a are formatted by concatenating the string that encodes $l(a)$ with a itself, followed by organizing the resulting string in chunks of 16-byte blocks. If necessary, the last block should be padded with zeros so that its length achieves 16 bytes. The blocks so constructed are appended to the first block B_0.

Message blocks are added right after the (optional) authentication blocks a. Message blocks are formatted by splitting the message P into 16-byte blocks and then padding the last block with zeros if necessary. If the message P consists of the empty string, then no blocks are added in this step. Then a sequence consisting of the concatenation of the blocks $B_0, B_1, \ldots, B_m$ is produced. Finally, the CBC–MAC is computed as

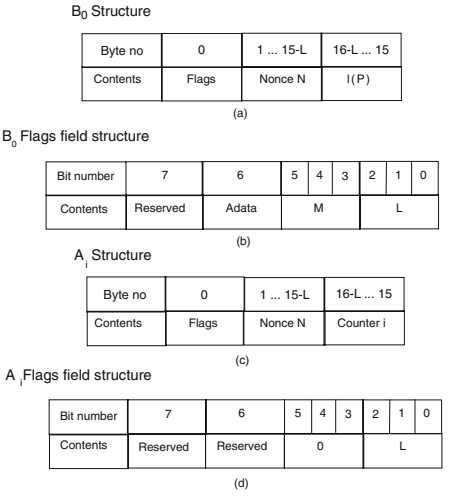

Fig. 12.8 Structure of the B_0 block and its flags.

$$\begin{aligned} X_1 &:= AES_E(K, B_0) \\ X_{i+1} &:= AES_E(K, X_i \oplus B_i) \text{ for } i = 1, \ldots, m \\ T &:= firstMbytes(X_{m+1}) \end{aligned} \tag{12.11}$$

where AES_E is the AES block cipher selected for encryption and T is the MAC value defined above. Note that the last block B_m is XORed with X_m and the result is encrypted with the block cipher. If it is needed, the ciphertext would be truncated in order to obtain T.

Figure 12.9 shows CCM authentication and verification processes dataflow. We stress that because of the CBC feedback nature of the CCM mode, we cannot use a pipeline approach when implementing a hardware CCM architecture.

12.8.1.3 CCM Encryption

CCM encryption is achieved by means of counter (CTR) mode as

$$\begin{aligned} S_i &:= AES_E(K, A_i) \text{ for } i = 0, 1, 2, \ldots, m \\ C_i &:= S_i \oplus P_i \end{aligned} \tag{12.12}$$

Figure 12.8c shows how the values A_i are formatted, where i is encoded in most-significant-byte first order. Within each block A_i, the Flags field is formatted as shown in Figure 12.8d. Once again, *reserved bits* field is reserved for future expansions and must be set to zero.

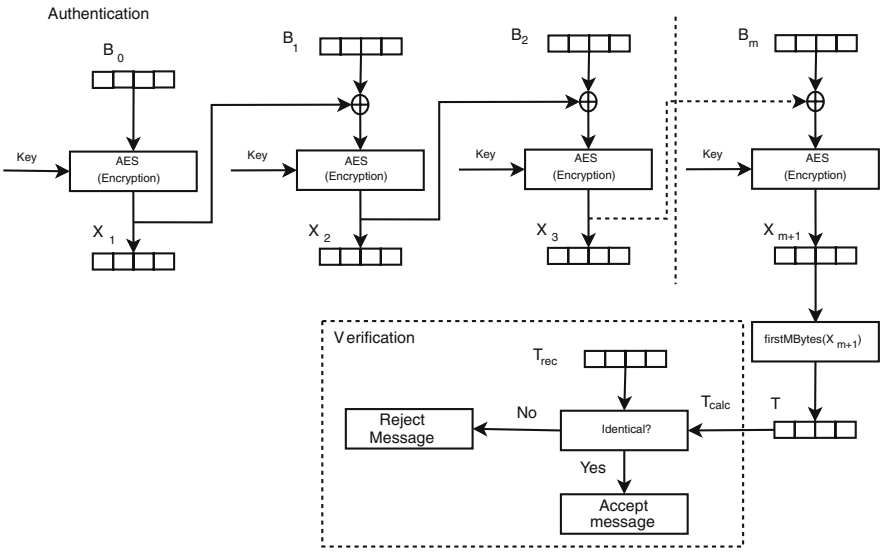

Fig. 12.9 Authentication and verification processes for the CCM mode.

Plaintext P is encrypted by XORing each of its bytes with the first $l(P)$ bytes of the sequence produced by concatenating the cipher blocks $S_1, S_2, S_3, \ldots$ produced by Equation 12.12. Notice that S_0 is not used for message encryption. The authentication value is computed by encrypting T with the key stream block S_0 truncated to the desired length as

$$U := T \oplus firstMbytes(S_0) \tag{12.13}$$

The final result C consists of the encrypted message P, followed by the encrypted authentication value U.

12.8.1.4 Decryption and Verification

To decrypt a message the following informations are required:

- The encryption key K
- The nonce N
- The additional authenticated data a
- The encrypted and authenticated message C

Decryption starts by recomputing the key stream to recover the message P and the MAC value T. Message and additional authentication data are then used to recompute the CBC–MAC value and check T.

If the T value is not correct, the receiver should not reveal the decrypted message, the value T or any other information.

It is important to notice that the AES encryption algorithm is required in both encryption as well as in decryption. Therefore, AES decryption functionality is not necessary in CCM mode, which results in valuable hardware resources saving.

Figure 12.10 shows the CCM encryption/decryption process dataflow.

12.8.2 AES Encryptor Core Implementation

As was mentioned before, in order to implement the CCM scheme, a 128-bit block cipher is needed. In this section we describe the general architecture of an AES sequential encryptor core as shown in Figure 12.11.

12.8.2.1 Implementation of the AES Rounds

Main nine rounds of AES must be implemented in an iterative way. Therefore, only one round is shown in Figure 12.12. That circuit uses a multiplexor to select whether we are going to process the first round or the other eight ones. At the end of the circuit we use a latch block to store the current computed state matrix.

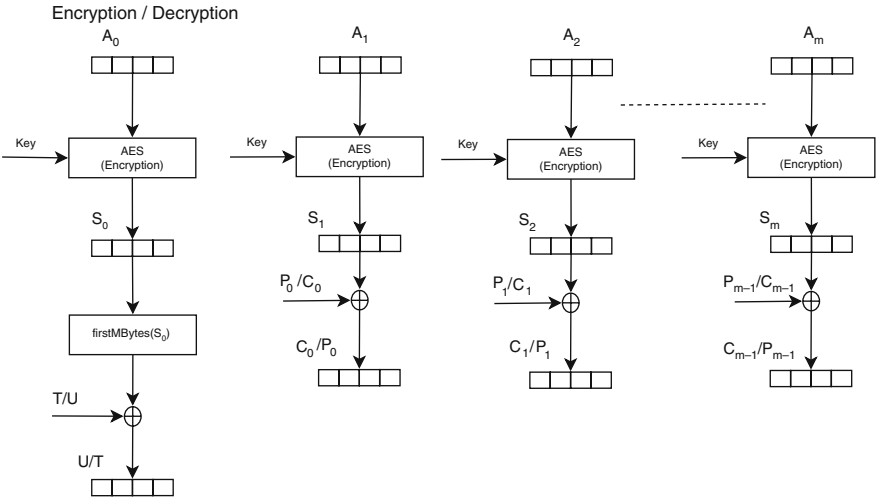

Fig. 12.10 Encryption and decryption processes for the CCM mode.

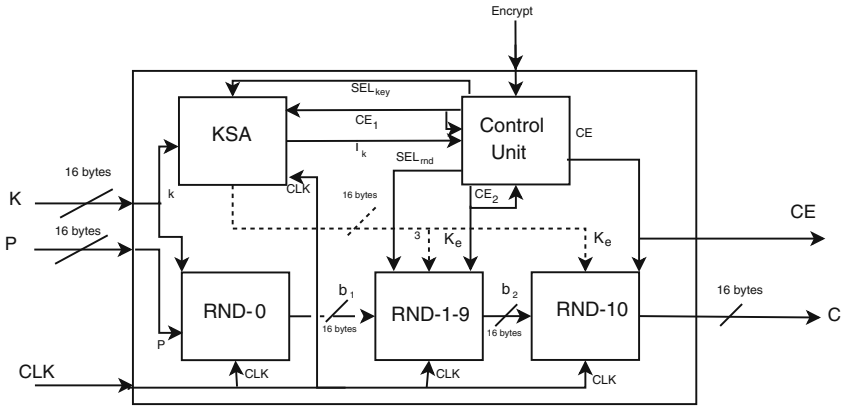

Fig. 12.11 General architecture of an AES encryptor core.

As is shown in Figure 12.12, rounds 1–9 were implemented using two main building blocks. The first one is the BS/SR block that can be instrumented by using the BRAMs (Block RAMs) embedded in the targeted FPGA device. Sixteen 8×256 BRAMs were configured for implementing AES S-box as a look-up table. By doing so, it is possible to compute 16-byte substitutions at the same time. Mix Columns and AddRoundKey Steps can be implemented jointly by doing some minor modifications. For polynomial multiplication the $xtime(v)$ operation described at the end of Section 12.4 was used.

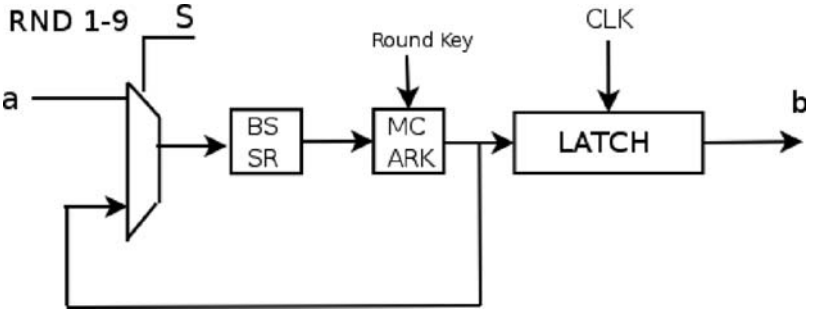

Fig. 12.12 Block diagram of the rounds 1–9.

12.8.2.2 Key Scheduling Implementation

In [47] several optimizations based on redundant computation for parallelizing the Key Scheduling process were implemented. As a result, it takes two steps to compute the round key [47].

Figure 12.13 shows Key Scheduling algorithm block diagram for an Iterative Encryptor Core. That circuit has a multiplexor that selects whether the key to be processed is the original user secret key or the current round key. At the output of the circuit we use a latch that stores the round key so produced. That key will be available until a new round key is generated. The latch is activated in the falling edge of the clock and its CE is activated in high state. The implementation shown can provide a round key every falling edge of the master clock.

12.8.2.3 AES Control Unit

The AES control unit synchronizes the whole process and controls the information flow. In addition, it produces the signals to control the multiplexors and latches that are used in the AES components. These synchronization signals are crucial because each component should select the correct state matrix.

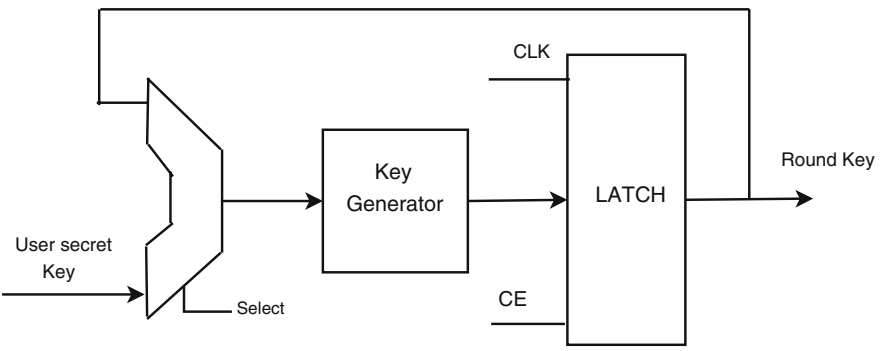

Fig. 12.13 Iterative key scheduling block diagram.

The signal generated to control the Final Round Latch is also used as an indicator that the ciphertext is ready. This is done by a change in the CE output of the AES block. When the plaintext is being processed, the CE output value is "0", but when a ciphertext is ready, this value changes to "1". In addition, the AES block has an extra input called "Encrypt" that indicates to the control unit that a new plaintext is given and that a new process has to begin. This control signal must be high by one single CLK's cycle and after that it must be set to low.

12.8.3 Hardware Implementation of the CCM Mode

In this section we discuss design details utilized for CCM mode and AES encryptor core implementations. It is assumed that the user must provide the additional authentication data *a* as two blocks of 16 bytes each (see Section 12.8.1). This size was selected considering the typical length of a TCP/IP header information. Furthermore, it was assumed that the message *P* to be processed has a maximum length of 1024 bytes.

Figure 12.14 shows the CCM mode general architecture, which comprises three main building modules, namely, authentication module, encryption module and a control unit module. All those three blocks together perform necessary operations for generating a valid cipher text and an encrypted authentication value. Notice that extra hardware is needed for the verification and decryption phases. In the rest of this section we will explain how those three blocks were implemented in [34].

12.8.3.1 CCM Authentication

Figure 12.15a depicts the CCM authentication module architecture. This module consists of an authentication block generator, a CBC–MAC module and a control unit.

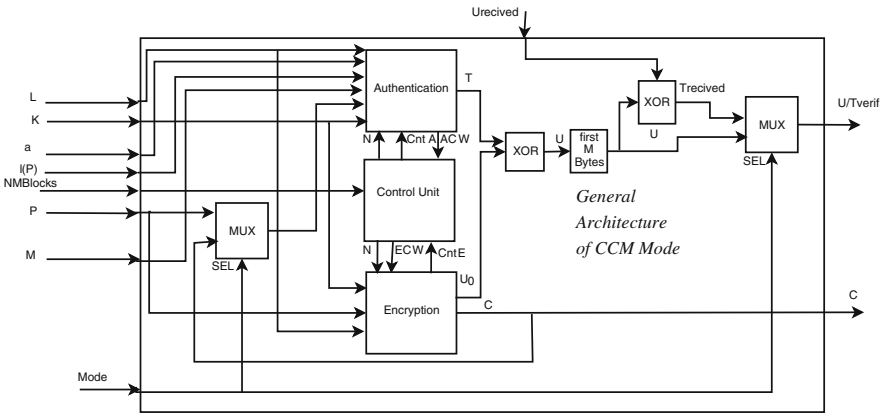

Fig. 12.14 CCM mode general architecture.

Authentication Block Generator: Authentication block generator is the architecture component responsible for generating the B_i blocks (see [15, 43] for details). Those blocks are generated according to the instructions indicated in the control word that the control unit sends in the "CW" line. That control word stipulates which block should be generated, the B_0 block or the blocks that correspond to the additional data a or the ones corresponding to the message P. Each block is generated only when the previous block has already been ciphered by the CBC–MAC module described next.

CBC–MAC: Blocks B_i that were generated by the block generator are the inputs for the CBC–MAC. Any input block B_i (except for the block B_0) is XORed with the X_i that was computed previously. The result of this operation is encrypted using AES, and the resulting cipher text X_{i+1} is fed back to the next block B_{i+1}. Figure 12.15b depicts the CBC–MAC process just outlined.

Authentication Control Unit: Control unit orchestrates the authentication process by receiving control signals from the general control unit. This block generates the appropriate control word for the block generator module and the one that indicates to the CBC–MAC Encryptor that a new B_i block can be processed. A 5-bit control word is utilized, where the LSB is used for controlling the CBC–MAC's latch. The second bit is used to start encryption with the AES block; the third bit controls which input will be selected by the MUX included in the CBC–MAC component and finally, the last two bits indicate which type of block should be generated.

Authentication control unit receives a signal when a B_i block has been processed within the CBC–MAC module. Thereafter, the control unit module produces the appropriate control word to generate the next block B_{i+1} and process it. Control unit runs a counter that indicates which control word should be generated. The process of authentication begins when the general control unit indicates so with the "ACW" word. After receiving this signal, the whole process is controlled by the local control unit. With the aim of parallelizing the authentication and encryption processes, the authentication begins first.

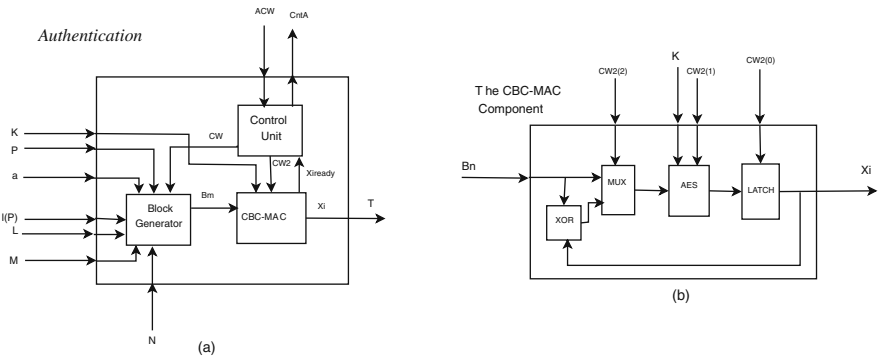

Fig. 12.15 Authentication block.

12.8.3.2 CCM Encryption

Figure 12.16a shows the CCM encryption architecture. This module consists of an encryption block generator, a CTR block and a control unit.

Encryption Block Generator: This module is responsible for generating the A_i blocks, (see Section 12.8.1 for details) generated according to the counter function. In this implementation, the counter begins in 0 and it is incremented one by one. The blocks are formed with the nonce and the counter value. Each block is generated when the CTR module has finished processing the previous one. Let us recall that encryption begins only when the authentication module is processing the second block with the additional data a. Based on this observation, the process of authentication and encryption can be accomplished in parallel by generating the A_1 block first and all subsequent A_i blocks but the first one (A_0). When the CCM authentication module has finished the last block processing, encryption block generator may proceed to generate the A_0 block in order to get S_0.

The CTR Mode: The CTR mode is the last step in the encryption process, it encrypts the A_i blocks with the block cipher (i.e., AES) to generate the S_i stream blocks. When a S_i block is ready, it is XORed with the appropriate message m block. Figure 12.16b shows the internal composition of the CTR mode. This mode uses as cipher block the AES implementation described in Section 12.8.2. The U_0 value shown in Figure 12.16b corresponds to the S_0 block. That is why U_0 is not XORed with the message, when it is the last block (actually the first of the counter function), this value is used to encrypt the authentication value T computed as a part of the authentication process as shown in Figure 12.9.

Encryption Control Unit: The implementation of this module is quite similar to the one for the authentication process. This control block is responsible to keep the counting and to tell the block generator which block is the next to be generated. At the same time, it starts the CTR mode for processing a block in order to get a valid ciphertext. When a ciphertext is ready, it tells to the general control unit that a new ciphertext can be stored. The implementation is based on the counter process that begins when the general control unit indicates that it is time to encrypt the message

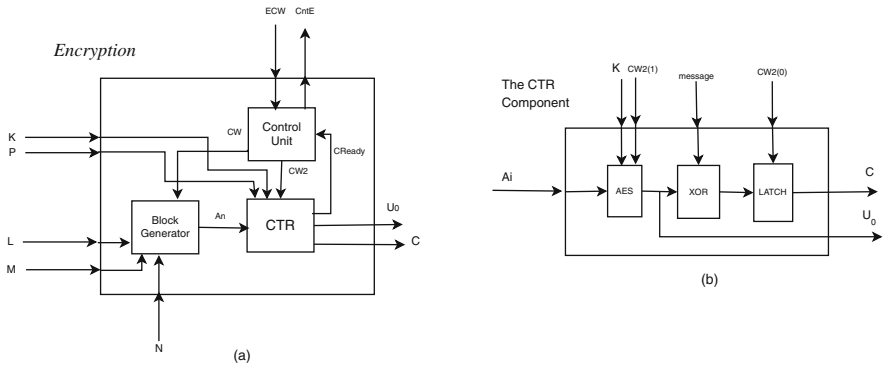

Fig. 12.16 Encryption block.

m, and it keeps counting until the general control unit indicates to stop. The control unit receives a control word that indicates what to do, this control word is 2-bits long, and the four possibilities are "00" or "10" do nothing, remain in initial state, "01" generate the S_0 block and "11" begin and continue counting.

12.8.3.3 General Control Unit

This module is the one that controls the authentication and encryption processes. It synchronizes the information flow in order to parallelize the entire process and to achieve a good performance. The control unit commands when the authentication process must begin the execution. After the authentication process has processed the first block (B_0) and continues with the second of the two additional data a blocks, the encryption process begins.

Notice that the authentication process must authenticate the other a block and all message blocks. The encryption process, on the other hand, must encrypt only the message blocks and since it starts first, one could think that the encryption process would finish first. However, since it is necessary for the extra processing of the block S_0, the architecture discussed here manages to finish both processes at the same time. In this way we can compute the encrypted authentication data U which is done with extra hardware after the authentication and encryption processes. Figure 12.17 shows this process time line. Every unit in the time line represents 12 clock cycles.

The control unit behaves in the same way for all, the authentication and encryption processes and for the decryption and verification processes, which means that the control unit does not know if it is encrypting or verifying.

12.8.3.4 Decryption and Verification

Decryption and verification processes are included in the architecture presented in [34] and they were implemented as extra hardware. The additional hardware is used

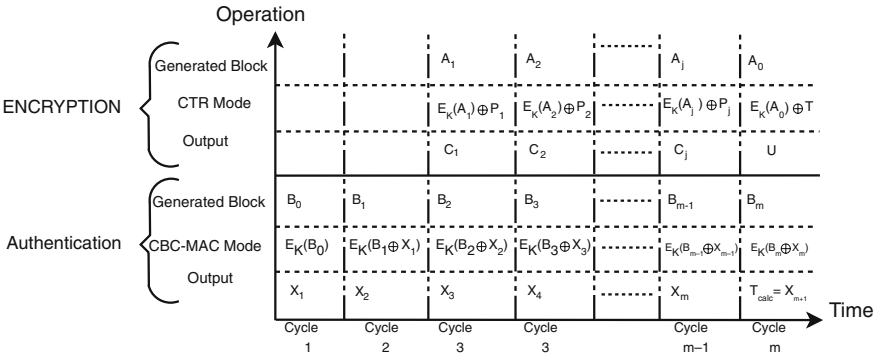

Fig. 12.17 Encryption–authentication time line.

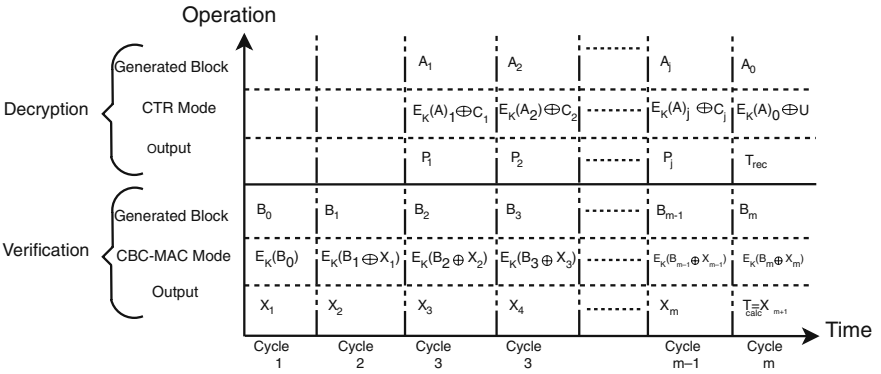

Fig. 12.18 Decryption–verification time line.

to select if the data are going to be authenticated and encrypted or decrypted and verified. In Figure 12.14 that extra hardware as a MUX is shown before the authentication process. This MUX is used to select if the source message is the one provided by the user (in case of authentication) or if it is the message that has been decrypted (when verifying). This is selected by the mode input, when "0" the process is going to authenticate and encrypt. If mode="1", then it will decrypt and verify.

The second MUX is used for selecting the output for the U value; when authenticating, the selection should be the computed value U, this is done with the function *firstMBytes* and the XOR operation between the computed T and the S_0 block. When verifying, computed U value is XORed with the U *received* (sent by the transmitter entity) in order to verify whether the message integrity has been corrupted or not; if the XOR output is equal to zero, then the message is correct, otherwise, it is assumed that the received message is corrupted.

As in the case of the authentication and encryption, the verification process begins first and the decryption process starts after the verification process has processed two blocks. In this way, verification can certify the received message that was just decrypted. Figures 12.17 and 12.18 show the time line of these processes, every unit in time line represents 12 clock cycles.

12.8.4 Experimental Results and Comparison

The design presented in this chapter (including key schedule) occupies 2154 slices, while it makes use of 32 block RAMs. It was implemented on a 3s4000fg900-4 Spartan 3 device using VHDL language and Xilinx's ISE 6.3i development tool along with the ModelSim Xilinx Edition II v5.8c [34]. Table 12.2 summarizes the hardware resources required by the design's main building blocks.

As was mentioned before, it was considered that the additional information a consisted of two 16-byte blocks. The maximum length of the plaintext is 1024 bytes which results in 64 blocks of 16 bytes each. Then, a total of 67 effective blocks must be processed (this estimation excludes the B_0 block, which was considered as

Table 12.2 Hardware resources of the design described in Section 12.8.3.

Block	Slices	BRAMs
Authentication	1031	16
Encryption	713	16
Control unit and extra Hw	410	0
CCM mode	2154	32
Maximum clock frequency	100.08 MHz	

Table 12.3 AES–CCM comparison.

Author	Device	Mode	Slices (BRAMs)	T* (Mbps)
AES–CCM core Helion	Virtex 4	CCM	480 (5)	670
	Virtex 5	CCM	321 (0)	760
Fu et al. [18]	Virtex 2	CTR	2415 (NA)	1490
Charot et al. [13]	Altera APEX	CTR	N/A	512
Bae et al. [3]	Altera Stratix	CCM	5605(LC)	285
This Design	Spartan 3	CBC	2154 (32)	1067

*Throughput

an overhead in the process, so it was omitted from the throughput computation). The processing of a total of 67 blocks can be accomplished by the design described in this section in 804 clock cycles (each block is computed in 12 cycles). Finally, we provide in Table 12.3 a comparison with several CCM–AES designs reported in the open literature.

12.9 Conclusions

Block ciphers are one of the most important symmetric cryptographic primitives. They are widely used for bulk encryptions. Block ciphers are always to be used along with an appropriate mode of operation when one needs to encrypt messages bigger than the block length of the block cipher. Also a mode of operation can provide security services other than privacy/confidentially. Thus, modes of operations are important cryptographic objects. A modern mode needs to be secure in terms of strong security definitions and also needs to be efficient in various respects. These two goals are sometimes contradictory and thus designing an efficient mode which is also secure is a challenging task.

From the beginning of this century many researchers have provided with many secure and efficient designs, though not many of the proposed modes are actually being used in applications. The standardization efforts for modes for different

applications is still going on and we hope that within a few years more modes would be standardized and thus widely used in various applications.

Efficient implementation of modes is another very important aspect. As per hardware design, AES has seen many efficient implementations, but all of these implementations may not be the best for every mode. A mode of operation may contain objects other than the block ciphers like field multipliers, hash functions, etc. Also the data dependencies for different kinds of modes may be quite different. This demands specific implementations of the mode and in particular the block cipher. Not many efficient implementations of different modes have yet been reported in the literature. In fact there are many modes which have not yet been implemented and therefore no test vectors are available for those modes.

Summarizing, in this chapter we provided a brief overview of hardware implementation aspects of modes of operations. We informally defined the various security goals for various modes, provided a partial list of different secure modes and described in detail the hardware implementation aspects of a secure mode called CCM.

Acknowledgments The authors gratefully acknowledge the valuable participation of Emmanuel López-Trejo in the AES–CCM hardware design described in this paper. Authors also acknowledge support from CONACyT through the CONACyT project number 60240-J1.

12.10 Exercises

1. In this chapter we showed that CBC mode is insecure if the IV is repeated. We also showed how to fix the problem. Show that the same is true for counter mode.
2. Assume that a plaintext of m blocks has been encrypted using a mode M. During transmission the $\frac{m}{2}$th block gets corrupted (assume $2|m$). Discuss which of the plaintext blocks will be corrupted after decryption, assuming the mode M to be ECB, CBC, CFB, OFB and CTR.
3. EME has a message length restriction, i.e., if the blocklength of the underlying block cipher is n it can securely encrypt only n blocks of message. Can you figure out why? (Hint: This has something to do with intermediate masking. You can look up the solution in [24].)

12.11 Projects

1. Implement any of the authenticated encryption modes in hardware. Provide test vectors for the mode selected.
2. Implement any of the disk encryption modes in hardware. Provide test vectors for the mode selected.

References

1. American National Standard for Financial Services X9.52-1998. *Triple Data Encryption Algorithm Modes of Operation*. American Bankers Association, Washington, D.C., July 1998.
2. B. Schneier. *Applied Cryptography*. Wiley, Second edition, 1996.
3. D. Bae, G. Kim, J. Kim, S. Park, and O. Song. An Efficient Design of CCMP for Robust Security Network. In *International Conference on Information Security and Cryptology*, vol. 3935, pp. 337–346, Seoul, Korea, Springer-Verlag, December 2005.
4. M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97*, pp. 394–403, Miami Beach, Florida, 1997.
5. M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *ASIACRYPT '00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*, pp. 531–545, London, UK, Springer-Verlag, 2000.
6. M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. In *FSE*, LNCS vol. 3017, pp. 389–407. Springer, 2004.
7. G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient software implementation of AES on 32-bits platforms. In *Proceedings of the CHES 2002*, LNCS vol. 2523 pp. 159–171. Springer, 2002.
8. D. Canright. A very compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, LNCS, vol. 3659 pp. 441–455. Springer, 2005.
9. A. Canteaut and K. Viswanathan, editors. *Progress in Cryptology – INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20–22, 2004, Proceedings*, LNCS, vol. 3348. Springer, 2004.
10. D. Chakraborty and P. Sarkar. A general construction of tweakable block ciphers and different modes of operations. In H. Lipmaa, M. Yung, and D. Lin, editors, *Inscrypt*, LNCS, vol. 4318 pp. 88–102. Springer, 2006.
11. D. Chakraborty and P. Sarkar. A new mode of encryption providing a tweakable strong pseudo-random permutation. In M. J. B. Robshaw, editor, *FSE*, LNCS, vol. 4047, pp. 293–309. Springer, 2006.
12. D. Chakraborty and P. Sarkar. HCH: A new tweakable enciphering scheme using the Hash-Encrypt-Hash approach. In R. Barua and T. Lange, editors, *INDOCRYPT*, LNCS vol. 4329, pp. 287–302. Springer, 2006.
13. F. Charot, E. Yahya, and C. Wagner. Efficient modular-pipelined AES implementation in counter mode on ALTERA FPGA. In P. Y. K. Cheung, G. A. Constantinides, and J. T. de Sousa, editors, *FPL*, LNCS, vol. 2778, pp. 282–291, Springer, 2003.

14. W. Diffie and M. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67:397–427, 1979.
15. D. Whiting, R. Housley, and N. Ferguson. Submission to NIST: Counter with CBC-MAC (CCM) AES mode of operation. Computer Security Division, Computer Security Resource Center (NIST), available at: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ccm/ccm.pdf
16. FIPS Publication 46-3. *Data Encryption Standard DES*. US DOC/NIST, October 1999.
17. FIPS Publication 81. *DES Modes of Operation*. US DOC/NIST, December 1980.
18. Y. Fu, L. Hao, and X. Zhang. Design of an extremely high performance counter mode AES reconfigurable processor. In *Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05)*, pp. 262–268. IEEE Computer Society, 2005.
19. B. Gladman. The AES Algorithm (Rijndael) in C and C++, available at: http://fp.gladman.plus.com/cryptography_technology/rijndael/
20. V. D. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. In M. Matsui, editor, *FSE 2001*, LNCS, vol. 2355, pp. 92–108. Springer, 2001.
21. T. Good and M. Benaissa. AES on FPGA from the fastest to the smallest. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 – September 1, 2005, Proceedings*, LNCS, vol. 3659, pp. 427–440. Springer, 2005.
22. S. Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. In Canteaut and Viswanathan, editors, *Progress in Cryptology – INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20–22, 2004, Proceedings*, LNCS, vol. 3348, pp. 315–327, Springer, 2004.
23. S. Halevi. TET: A wide-block tweakable mode based on Naor-Reingold. Cryptology ePrint Archive, Report 2007/014, 2007. http://eprint.iacr.org/
24. S. Halevi and P. Rogaway. A tweakable enciphering mode. In D. Boneh, editor, *CRYPTO*, LNCS vol. 2729, pp. 482–499. Springer, 2003.
25. S. Halevi and P. Rogaway. A parallelizable enciphering mode. In T. Okamoto, editor, *CT-RSA*, LNCS vol. 2964, pp. 292–304. Springer, 2004.
26. S. F. Hsiao and M. C. Chen. Efficient substructure sharing methods for optimising the inner-product operations in Rijndael Advanced Encryption Standard. *IEE Proceedings on Computer and Digital Technology*, 152(5):653–665, September 2005.
27. T. Ichikawa, T. Kasuya, and M. Matsui. Hardware evaluation of the AES finalists. In *The Third AES3 Candidate Conference*, pp. 279–285, New York, April 2000.
28. J. Daemen and V. Rijmen. *The Design of Rijndael: AES The Advanced Encryption Standard*. Springer-Verlag, First edition, 2002.

29. C. S. Jutla. Encryption modes with almost free message integrity. Cryptology ePrint Archive, Report 2000/039, 2000. http://eprint.iacr.org/
30. C. S. Jutla. Encryption modes with almost free message integrity. In B. Pfitzmann, editor, *EUROCRYPT*, LNCS, vol. 2045, pp. 529–544. Springer, 2001.
31. T. Kohno, J. Viega, and D. Whiting. CWC: A high-performance conventional authenticated encryption mode. Cryptology ePrint Archive, Report 2003/106, 2003. http://eprint.iacr.org/
32. H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES Modes of Operations: CTR-mode encryption, September 2000, available at: http://www.cs.ucdavis.edu/rogaway/papers/ctr.pdf.
33. M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, pp. 31–46, London, UK, Springer-Verlag, 2002.
34. E. López-Trejo, F. R. Henríquez, and A. Díaz-Pérez. An Efficient FPGA Implementation of CCM mode using AES. In *International Conference on Information Security and Cryptology*, LNCS, vol. 3935, pp. 208–215, Seoul, Korea, Springer-Verlag, December 2005.
35. M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal of Computing*, 17(2):373–386, 1988.
36. A. K. Lutz, J. Treichler, F. K. Gurkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fitchtner. 2 Gbits/s Hardware realization of RIJNDAEL and SERPENT-A comparative analysis. In *Proceedings of the CHES 2002*, LNCS, vol. 2523, pp. 171–184. Springer, 2002.
37. C. Mancillas-López, D. Chakraborty, and F. Rodríguez-Henríquez. Efficient implementations of some tweakable enciphering schemes in reconfigurable hardware. In K. Srinathan, C. P. Rangan, and M. Yung, editors, *INDOCRYPT*, LNCS, vol. 4859, pp. 414–424. Springer, 2007.
38. C. Mancillas-Lopez, D. Chakraborty, and F. Rodriguez-Henriquez. Reconfigurable hardware implementations of tweakable enciphering schemes. Cryptology ePrint Archive, Report 2007/437, 2007. http://eprint.iacr.org/
39. D. A. McGrew and S. R. Fluhrer. The Extended Codebook (XCB) mode of operation. Cryptology ePrint Archive, Report 2004/278, 2004. http://eprint.iacr.org/
40. D. A. McGrew and J. Viega. Arbitrary block length mode, 2004, available at: http://grouper.ieee.org/groups/1619/email/pdf00005.pdf
41. D. A. McGrew and J. Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In Canteaut and Viswanathan, editors, *Progress in Cryptology – INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20–22, 2004, Proceedings*, LNCS, vol. 3348, pp. 343–355. Springer, 2004.
42. M. Dworkin. Recommendation for Block Cipher Modes of operation methods and techniques. National Institute of Standards and Technology (NIST), December 2001 available at: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

43. M. Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for authentication and confidentiality. National Institute of Standards and Technology (NIST), May 2004, available at: http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf
44. M. Naor and O. Reingold. A pseudo-random encryption mode. Manuscript available at: www.wisdom.weizmann.ac.il/naor
45. M. Naor and O. Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology*, 12(1):29–66, 1999.
46. NIST Special Publication 800-38A 2001 edition. *Recommendation for Block Cipher Modes of Operation*. US NIST, December 2001.
47. F. Rodríguez-Henríquez, N. A. Saqib, and A. Díaz-Pérez. 4.2 Gbit/s single-chip FPGA implementation of AES algorithm. *IEE Electron. Lett.*, 39(15):1115–1116, July 2003.
48. P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In P. J. Lee, editor, *ASIACRYPT*, LNCS, vol. 3329, pp. 16–31. Springer, 2004.
49. P. Rogaway. Nonce-based symmetric encryption. In B. K. Roy and W. Meier, editors, *FSE*, LNCS vol. 3017, pp. 348–359. Springer, 2004.
50. P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transaction on Information Systems Security*, 6(3):365–403, 2003.
51. A. Rudra, P. K. Dubey, C. S. Julta, V. Kumar, J. R. Rao, and P. Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In *Proceedings of the CHES 2001*, LNCS, vol. 2162, pp. 171–184. Springer, 2001.
52. P. Sarkar. Improving upon the TET mode of operation. In K.-H. Nam and G. Rhee, editors, *ICISC*, LNCS, vol. 4817, pp. 180–192. Springer, 2007.
53. W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, First edition, 2002.
54. P. Wang, D. Feng, and W. Wu. HCTR: A variable-input-length Enciphering mode. In D. Feng, D. Lin, and M. Yung, editors, *CISC*, LNCS, vol. 3822, pp. 175–188. Springer, 2005.

Chapter 13
Basics of Side-Channel Analysis

Marc Joye

13.1 Introduction

Classical cryptography considers attack scenarios of adversaries getting black box access to a cryptosystem, namely to its inputs and outputs. For example, in a chosen ciphertext attack, an adversary can submit ciphertexts of her choice to a decryption oracle and receives in return the corresponding plaintexts. In real life, however, an adversary may be more powerful. For example, an adversary may in addition monitor the execution of the cryptosystem under attack and collect some side-channel information, such as the execution time or the power consumption. The idea behind side-channel analysis is to infer some secret data from this extra information.

This chapter presents several applications of side-channel analysis using different types of side-channel leakage. The primary goal is to explain the basic principles of side-channel analysis through concrete examples. Simple countermeasures to prevent side-channel leakage are also discussed. More sophisticated methods and advanced techniques are presented in the next chapters.

13.2 Timing Analysis

The concept of using *side-channel information* as a means to attack cryptographic schemes first appeared in a seminal paper by Kocher [8]. In this paper, Kocher exploits differences in computation times to break certain implementations of RSA and of discrete logarithm-based cryptosystems.

In this section, we describe two timing attacks. We show how an attacker able to make timing measurements with some accuracy can recover secret data. The first attack applies to a password verification routine. Following [5], the second attack is against an implementation of an RSA signature scheme [2, 13].

Thomson R&D France
e-mail: marc.joye@thomson.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_13,
© Springer Science+Business Media, LLC 2009

13.2.1 Attack on a Password Verification

In order to restrict access to data to legitimate users, certain applications make use of passwords. Think for example of a file that requires a valid password to be opened. If the password is an 8-byte value, a brute-force attack would require $256^8 = 2^{64}$ trials!

Now assume that the password verification routine is implemented as described in Algorithm 4.[1] $\widetilde{P}$ denotes the 8-byte password proposed by the user and P denotes the correct password. The routine returns "`true`" if the entered password is valid and "`false`" if it is not.

Algorithm 4 Password verification.

Input: $\widetilde{P} = (\widetilde{P}[0], \ldots, \widetilde{P}[7])$ (and $P = (P[0], \ldots, P[7])$)
Output: '`true`' or '`false`'
1: **for** $j = 0$ to 7 **do**
2: **if** $(\widetilde{P}[j] \neq P[j])$ **then return** '`false`'
3: **end for**
4: **return** '`true`'

This implementation is insecure against an attacker measuring the time taken by the routine to return the status, "`true`" or "`false`". We can see that the verification of a valid password is longer since the routine returns "`false`" as soon as two bytes differ. Based on this, an attacker can mount the following attack.

1. For $0 \leq n \leq 255$, the attacker proposes the 256 passwords $\widetilde{P}^{(n)} = (n, 0, 0, 0, 0, 0, 0, 0)$ and measures the corresponding running time, $\tau[n]$.
2. Next, the attacker computes the maximum running time

$$\tau[n_0] := \max_{0 \leq n \leq 255} \tau[n] \ .$$

   The correct value for the first byte of P, $P[0]$, is given by n_0.
3. Once $P[0]$ is known, the attacker reiterates the attack with

$$\widetilde{P}^{(n)} = (P[0], n, 0, 0, 0, 0, 0, 0),$$

   and so on until the whole value of P is recovered.

This timing attack is very efficient. It requires at most $256 \cdot 8 = 2048$ calls to the verification routine to completely recover an 8-byte password.

To prevent the attack, one may think of adding a random delay before returning the status. This is however not sufficient since the attacker would still be able to mount a similar attack. In Step 1, the attacker would propose t times the password

[1] Note that this is basically the way the C function `memcmp` compares two memory regions.

$\widetilde{P}^{(n)} = (n,0,0,0,0,0,0,0)$ and measure the corresponding average running time over the t executions, $\bar{\tau}[n]$, for $0 \leq n \leq 255$. The attack then proceeds as before. Its complexity increases by a factor of t.

The correct way to prevent the attack is to make a *constant time* implementation irrespective of the input data.

13.2.2 Attack on an RSA Signature Scheme

Like a handwritten signature, the purpose of a digital signature is to guarantee the authenticity and the integrity of a message. There are two keys: a signing key which is private and a verification key which is public.

A common practice to produce a digital signature of a message m with RSA relies on the "hash-and-sign" paradigm. Message m is first hashed into $\mu(m)$ and the result is then raised to the dth power modulo N, $S = \mu(m)^d \bmod N$, where d denotes the private RSA key. The public verification key is $\{e,N\}$ where $ed \equiv 1 \pmod{\phi(N)}$ and ϕ is Euler's totient function. The validity of a putative signature S of message m is verified by checking whether $S^e \equiv \mu(m) \pmod N$.

There are various ways to evaluate a modular exponentiation. We consider below the square-and-multiply algorithm. At iteration j of the main loop, there is a modular squaring and when bit d_j of d is equal to 1 there is also a modular multiplication.

Algorithm 5 Computation of an RSA signature.

Input: $m, N, d = (d_{k-1}, \ldots, d_0)_2$, and $\mu : \{0,1\}^* \to \mathbb{Z}/N\mathbb{Z}$
Output: $S = \mu(m)^d \pmod N$
1: $R_0 \leftarrow 1; R_1 \leftarrow \mu(m)$
2: **for** $j = k-1$ downto 0 **do**
3: $\quad R_0 \leftarrow {R_0}^2 \pmod N$
4: $\quad$ **if** $(d_j = 1)$ **then** $R_0 \leftarrow R_0 \cdot R_1 \pmod N$
5: **end for**
6: **return** R_0

For better performance, the modular multiplications can be evaluated using Montgomery method [10]. Montgomery modular multiplication produces a result correct modulo N but lying in the interval $[0,2N[$ so that a subtraction by N may be needed to have the result in $[0,N[$ as expected.[2]

As we will see, the time required to perform this possible subtraction can be discriminatory and allows an attacker to recover the value of secret exponent d. The attack iteratively recovers d, bit-by-bit, starting from the most significant position.

[2] We note that there are implementations of Montgomery multiplication that directly produce a result in $[0,N[$. See [6, 14].

We assume that the attacker already knows $d_{k-1},\ldots,d_{k-n+1}$. Her goal is to recover the value of the next bit, d_{k-n}.

1. The attacker *guesses* that $d_{k-n}=1$.
2. Next, the attacker randomly chooses t messages, $m_1,\ldots,m_t$, and prepares two sets of messages, $\mathscr{S}_0$ and $\mathscr{S}_1$, given by

$$\mathscr{S}_0=\big\{m_i \mid \text{Montgomery multiplication } R_0\leftarrow R_0\cdot R_1 \text{ in Line 4 of Algorithm 5 does not induce a subtraction for } j=k-n\big\}$$

   and

$$\mathscr{S}_1=\big\{m_i \mid \text{Montgomery multiplication } R_0\leftarrow R_0\cdot R_1 \text{ in Line 4 of Algorithm 5 induces a subtraction for } j=k-n\big\}\ .$$

3. For each message in set $\mathscr{S}_0$, the attacker requests the signature and measures the computation time to get it. She does the same for messages in set $\mathscr{S}_1$. Let $\bar{\tau}_0$ and $\bar{\tau}_1$ denote the average time (per signature request) for messages in $\mathscr{S}_0$ and $\mathscr{S}_1$, respectively.
4. If $\bar{\tau}_1\approx\bar{\tau}_0$ then the guess of the attacker was wrong and $d_{k-n}=0$. If $\bar{\tau}_1\gg\bar{\tau}_0$ (more precisely, if the time difference between $\bar{\tau}_1$ and $\bar{\tau}_0$ is roughly the time of a subtraction) then the attacker correctly guessed that $d_{k-n}=1$.
5. Now that the attacker knows $d_{k-1},\ldots,d_{k-n+1},d_{k-n}$, she iterates the attack to recover the value of d_{k-n-1} and so on.

It is worth noting that if $d_{n-k}=0$ then two sets, $\mathscr{S}_0$ and $\mathscr{S}_1$, behave as two random sets and so the average computation time for messages in $\mathscr{S}_0$ and $\mathscr{S}_1$ will be roughly the same.

The previous attack can be improved in a number of ways. In particular, the knowledge of public exponent $e=d^{-1} \bmod \phi(N)$ and lattice basis reduction techniques may help to speed up the recovery of d [4].

13.3 Simple Power Analysis

The power consumption of cryptographic devices, such as smart cards, depends on the manipulated data and of the executed instructions. It can be monitored on an oscilloscope by inserting a resistor in series with the ground or power supply pin. The so-obtained measurement is called a *power trace*.

Simple power analysis (SPA) [7] is a technique that involves direct interpretation of a power trace. This section presents some applications of simple power analysis. We show how to reverse-engineer the code of a program, namely how to recover an unknown instruction. The two other applications are actually attacks. We mount key recovery attacks against implementations of a private RSA exponentiation and a DES key schedule [11], as alluded in [8, Section 11].

13.3.1 Reverse-Engineering of an Algorithm

Several models are used to characterize the information leakage through the power consumption. The simpler model is the *Hamming-weight model*. It assumes that the power consumption varies according to the Hamming weight (i.e., the number of non-zero bits in a given bit string) of the manipulated data or the executed instruction.

Another model that works well in practice for many cryptographic devices is the *Hamming-distance model*. This model considers the number of flipping bits in the current state compared with the previous state. If Hw denotes the Hamming-weight function and if state_t and state_{t-1}, respectively, denote the state at clock cycles t and $t-1$ then the Hamming distance is given by

$$\mathrm{Hw}(\mathsf{state}_t \oplus \mathsf{state}_{t-1}),$$

where $\oplus$ denotes the XOR (exclusive OR) operator. It is easy to see that the above relation yields the number of flipping bits.

Of course, other models can be imagined for different technologies or chips.

Figure 13.1 shows four power traces from a given smart card as it evaluates $f(x,0)$ for byte values $x = 0, 1, 7$ and 255. We see that the traces are almost identical everywhere except at a few locations. These differences are called *trace signatures*. Different values for x give rise to different power consumption levels.

Trace signatures can be exploited to reverse-engineer a program code. Imagine that function f in the above example is performed on a smart card and is unknown. More precisely, imagine that we just know that at some point in the computation the smart card loads input byte x into the accumulator and applies some binary operator, say ∂, to x and 0:

$$f(x,0) = x \,\partial\, 0.$$

If the leakage model is known, it is easy to recover the unknown instruction ∂. Assume that the smart card follows the Hamming-distance model. It suffices to choose a location in the power trace that presents a signature and, for each input value $x \in [0,\ldots,255]$, to measure the corresponding power consumption. The measured

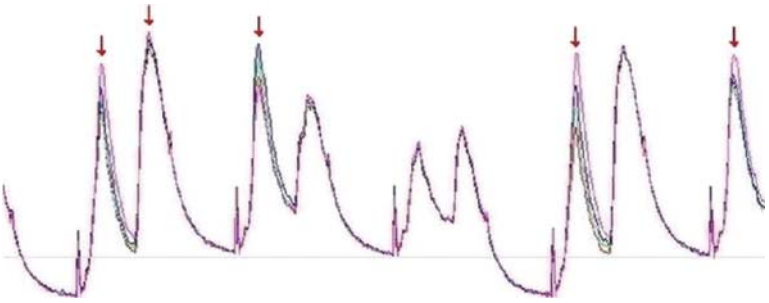

Fig. 13.1 Power traces of $f(x,0)$ for different values of x.

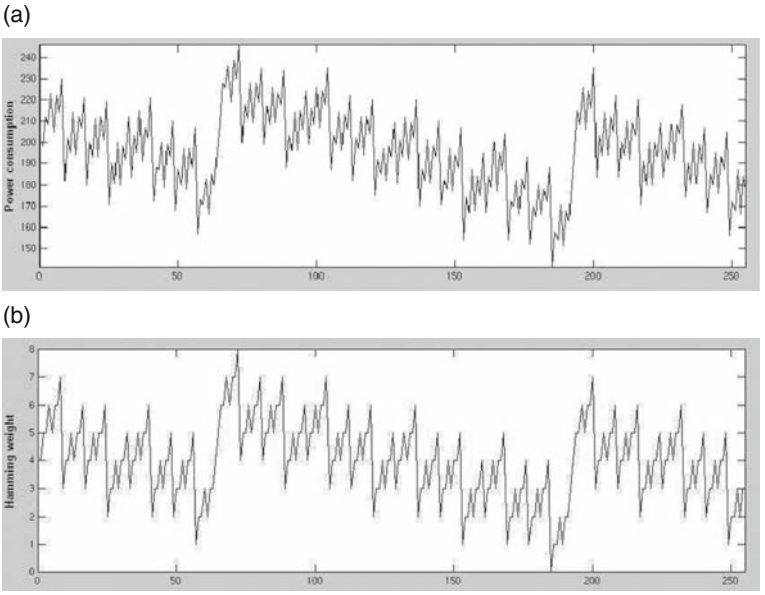

Fig. 13.2 Recovering an unknown instruction. (a) Instant power consumption, for $x \in [0,\ldots,255]$. (b) Hamming weight of $184 \oplus x$, for $x \in [0,\ldots,255]$.

power consumptions are then reported on a graph as depicted on Figure 13.2a. The next step is, for each byte value $\partial \in [0,\ldots,255]$, to plot the graph $\big(x, \mathrm{Hw}(\partial \oplus x)\big)$ for $x \in [0,\ldots,255]$. The graph that most resembles Figure 13.2a yields the value of unknown instruction. In our example, this unknown instruction is "184" which, for the considered smart card, corresponds to an XOR. Observe that Figure 13.2a and Figure 13.2b have globally the same shape.

13.3.2 Attack on a Private RSA Exponentiation

Figure 13.3 below represents (the beginning of) a power trace corresponding to the computation of an RSA signature with the square-and-multiply algorithm as per Algorithm 5.

We can distinguish two patterns in the power trace: a high-level pattern and a low-level pattern. We know that, at each step, the square-and-multiply algorithm performs a square and if the exponent bit is a 1 it also performs a multiply. Hence, it is not difficult to deduce that high levels correspond to multiplies and low levels to squares. The very beginning of the power trace presents a very low level of power consumption, this corresponds to the squares $R_0 \leftarrow 1^2 \pmod N$ (cf. Line 3 in Algorithm 5) until the first non-zero bit of d is encountered, in which case the accumulator, R_0, contains $\mu(m)$: $R_0 \leftarrow 1 \cdot R_1 \pmod N = \mu(m)$. The next bits of d are recovered from the left to the right with the following rule: A low level followed

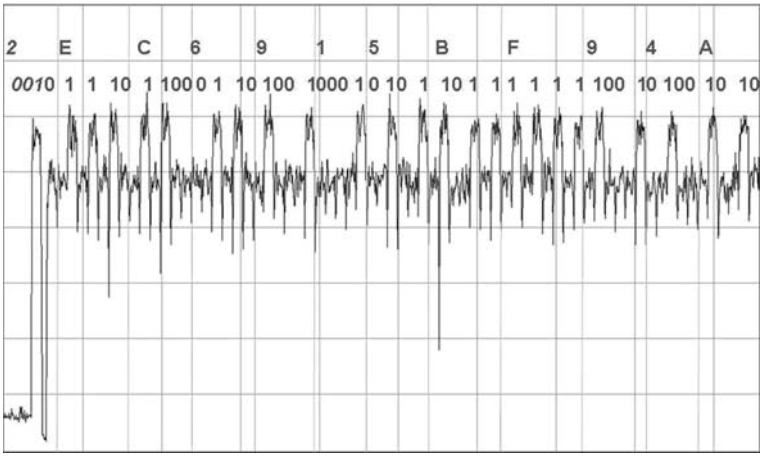

Fig. 13.3 Power trace of an RSA exponentiation.

by a high level corresponds to a 1-bit and a low level not followed by a high level corresponds to a 0-bit. So, we get (in hexadecimal) $d = \texttt{2EC6915BF94A}\ldots$, that is, the private RSA key!

Contrary to the timing attack of Section 13.2.2, this SPA attack equally applies to signature schemes using probabilistic paddings like RSA-PSS [3].

13.3.3 Attack on a DES Key Schedule

SPA-type attacks are not restricted to public-key algorithms but can potentially be applied to other types of cryptographic algorithms. Actually, a power trace can be viewed as a two-dimensional leakage function since it gives the power consumption level at a given time. It also reveals local timing information. We present in this section an SPA attack against an implementation of the DES key schedule.

DES is a block cipher that operates on 64-bit blocks of plaintext; the key length is 56 bits. DES comprises 16 identical rounds and makes use of a 48-bit sub-key for each round. The round sub-keys are obtained through what is called the key schedule. At round t, $1 \leq t \leq 16$, a 56-bit input buffer K_{t-1} is split into two 28-bit halves, C_{t-1} and D_{t-1}, and each half is circularly shifted by b bits to the left[3]—the value of b depending on the round number (see Table 13.1)—to produce a 56-bit output buffer K_t:

$$K_t = C_t \| D_t \quad \text{with } C_t = C_{t-1} \circlearrowleft b \text{ and } D_t = D_{t-1} \circlearrowleft b\,,$$

and where $\circlearrowleft$ denotes the circular shifting to the left.

[3] When DES is used in decryption, the circular shifting is done to the right.

Table 13.1 Number of key bits (b) shifted per round.

Round number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
DES	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1
DES^{-1}	0	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

The round sub-key is then obtained by extracting 48 bits from K_t. The initial value K_0 is derived from the cipher key.

Algorithm 6 describes an implementation of a DES key shifting by one bit to the left. The algorithm is called b times to shift by b bits to the left. As the 56-bit input buffer K_{t-1} is updated with the 56-bit output buffer K_t, we drop the subscript and consider a 56-bit buffer $K = C|D$. In this implementation, buffer K is coded on 7 bytes $K[0], K[1], \ldots, K[6]$ with $0 \leq K[i] \leq 255$. We let $(K[i]_7, K[i]_6, \ldots, K[i]_0)_2$ denote the binary representation of byte $K[i]$ $(0 \leq i \leq 6)$. So, we have $C = (K[0]_7, \ldots, K[0]_0, K[1]_7, \ldots, K[1]_0, K[2]_7, \ldots, K[2]_0, K[3]_7, \ldots, K[3]_4)_2$ and $D = (K[3]_3, \ldots, K[3]_0, K[4]_7, \ldots, K[4]_0, K[5]_7, \ldots, K[5]_0, K[6]_7, \ldots, K[6]_0)_2$,

$$
\begin{array}{cccccccc}
K[0]_7 & K[0]_6 & K[0]_5 & K[0]_4 & K[0]_3 & K[0]_2 & K[0]_1 & K[0]_0 \\
K[1]_7 & K[1]_6 & K[1]_5 & K[1]_4 & K[1]_3 & K[1]_2 & K[1]_1 & K[1]_0 \\
K[2]_7 & K[2]_6 & K[2]_5 & K[2]_4 & K[2]_3 & K[2]_2 & K[2]_1 & K[2]_0 \\
K[3]_7 & K[3]_6 & K[3]_5 & K[3]_4 & K[3]_3 & K[3]_2 & K[3]_1 & K[3]_0 \\
K[4]_7 & K[4]_6 & K[4]_5 & K[4]_4 & K[4]_3 & K[4]_2 & K[4]_1 & K[4]_0 \\
K[5]_7 & K[5]_6 & K[5]_5 & K[5]_4 & K[5]_3 & K[5]_2 & K[5]_1 & K[5]_0 \\
K[6]_7 & K[6]_6 & K[6]_5 & K[6]_4 & K[6]_3 & K[6]_2 & K[6]_1 & K[6]_0
\end{array}.
$$

Algorithm 6 DES key shifting.

1: $\texttt{carry} \leftarrow K[3]_3$
2: **for** $j = 6$ downto 0 **do**
3: $\quad K[j] \leftarrow K[j] \circlearrowleft 1$ ▷ This operation also affects the carry flag!
4: **end for**
5: $K[3]_4 \leftarrow 0$
6: **if** (carry) **then** $K[3]_4 \leftarrow 1$

At the end of the **for** loop, the carry contains the value of bit $K[0]_7$ before shifting. This bit should replace bit $K[3]_4$ after shifting. This is done in two steps: $K[3]_4$ is first forced to "0" and if the carry is set then it is written as "1". Because the corresponding power trace will present a different pattern depending on the input bit $K[0]_7$, the value of this bit can be deduced. Moreover, since after 16 rounds 28 bits are going into the carry in a DES encryption (see Table 13.1), the value of 28 cipher-key bits can be recovered. The value of the 28 remaining cipher-key bits can be found by exhaustive search or by applying the same attack on DES in decryption mode (which yields the value of 27 bits, see Table 13.1).

13.4 Differential Power Analysis

Differential power analysis (DPA) [7] is a sophisticated power analysis technique that makes use of statistical methods on a collection of power traces. This section introduces the important notions: the DPA selection function and the DPA trace. We show how this enables an attacker to locate an algorithm within a power trace. We also show how this can be used to recover secret keys of cryptographic algorithms. To illustrate this, we present a DPA attack against an implementation of the AES block cipher [12] and generalize the timing attack presented in Section 13.2.2.

13.4.1 Bit Tracing

It is not always easy to precisely locate things in a power trace. Imagine for example that somewhere in a long cryptographic process data are encrypted to form a ciphertext which is next transferred to another location of RAM memory. The goal is to locate the encryption algorithm.

We need to introduce some notation. Let σ denote a *Boolean selection function* which returns $\sigma(y) = 0$ or $\sigma(y) = 1$, depending on the value of y. Let also $\langle\cdot\rangle$ represent the average operator and $\mathscr{C}_P(t)$ denote the power consumption of process P at time period t.

Differential power analysis runs in two phases. In the first phase, several power traces of a same process P are collected. In the second phase, depending on a *known* intermediate value y, a partition of two sets, $\mathscr{S}_0$ and $\mathscr{S}_1$, is made:

$$\mathscr{S}_0 = \{y \mid \sigma(y) = 0\} \quad \text{and} \quad \mathscr{S}_1 = \{y \mid \sigma(y) = 1\} \ .$$

The *DPA trace* is then given by

$$\Delta_P(t) := \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_1} - \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_0} ,$$

namely, the difference of the average power consumption curve corresponding to sets $\mathscr{S}_1$ and $\mathscr{S}_0$, for each time period t.

Basically, the DPA trace magnifies the effect of selection function σ. Back to our example, suppose that the cryptographic process is performed on an 8-bit device that obeys the Hamming-weight model. Suppose further that the ciphertexts are known and are represented on 16 bytes. If selection function σ is defined as returning the value of a given bit of the first byte of the ciphertext then sets $\mathscr{S}_0$ and $\mathscr{S}_1$ will contain ciphertexts for which a given bit is always a "0" and a "1", respectively. As a consequence, the average Hamming-weight value for the first byte of ciphertexts in set $\mathscr{S}_0$ will be of 3.5 while for ciphertexts in set $\mathscr{S}_1$ its average value will be of 4.5. Furthermore, since in the Hamming-weight model the power consumption is a function of the Hamming weight of the manipulated data, this difference between the average Hamming weights will translate into a difference between the average

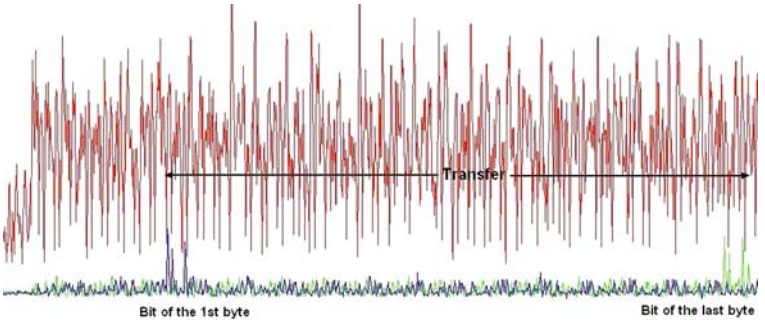

Fig. 13.4 Locating an algorithm.

power consumptions for set $\mathscr{S}_0$ and set $\mathscr{S}_1$ when the first byte of the ciphertext is being manipulated. This appears as a peak in the DPA trace. DPA peaks allow one to trace the bit used for the selection: each time this bit is manipulated a DPA peak should appear.

Applied to our example, we get a first DPA trace (bottom curve in Figure 13.4 presenting two peaks on the left-hand side). The second DPA trace (bottom curve in Figure 13.4 presenting two peaks on the right-hand side) is obtained similarly by the selection bit as a given bit in the last byte of the ciphertext. Hence, the ciphertext is manipulated between the two points determined by the DPA peaks. We can assume that this is the ciphertext being transferred for emission on the I/O or used in a subsequent function. In order for the ciphertext to exist, the encryption algorithm will have been executed. We therefore know that encryption algorithm is somewhere before in the power trace (top curve in Figure 13.4).

13.4.2 Attack on an AES Implementation

DPA can also be used to recover secret information. In the previous section, we have seen that DPA allows one to trace the activity of a selected bit. The idea was to use this bit to make a partition of two sets. An encryption algorithm takes on input a message and an encryption key to form a ciphertext. If the encryption key is unknown, it is not possible to make a partition on intermediate values during the course of the encryption. Nevertheless, DPA can be used by an attacker to validate the value of a key candidate as follows.

1. The attacker makes a guess on the value of the key.
2. Next, she applies the DPA methodology to some intermediate value depending on the key.
3. If the DPA trace does not present DPA peaks then it means that the guess was wrong and the attacker goes back to step 1. If it does present peaks then the attacker found the key, see, e.g., Figure 13.5.

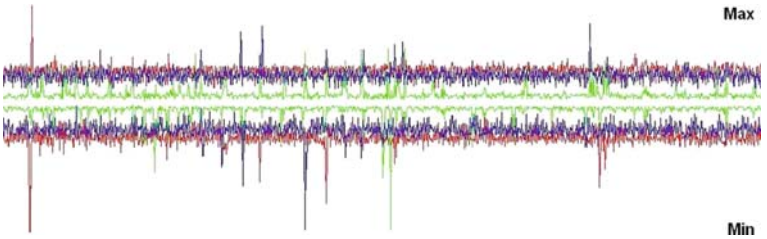

Fig. 13.5 DPA traces for different selection bits.

At first glance, it is not apparent to see in what this differs from a regular exhaustive key search. The difference is that the technique may be applicable to *part* of the key. We illustrate this on an implementation of the AES cipher.

The Advanced Encryption Standard (AES) is the successor of the older DES. The AES block cipher encrypts 128-bit blocks of plaintext and supports key lengths of 128, 192 and 256 bits. We consider the 128-bit version. The 128-bit plaintext, m_i, and 128-bit cipher key, K, are viewed as (4×4) matrices of bytes

$$m_i = (s_{u,v}^{(i)})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}} \quad \text{and} \quad K = (k_{u,v})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}}.$$

Plaintext m_i is first XORed with the cipher key and then gradually updated by applying round functions `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` in a series of 10 rounds. The `SubBytes` function substitutes each byte $s_{u,v}^{(i)}$, $0 \le u, v \le 3$, by another byte through a non-linear permutation $\mathrm{S_{RD}}$. As `SubBytes` has an effect on the value of a whole byte, selection function σ can be defined as the value of a given output bit of $\mathrm{S_{RD}}(s_{u,v}^{(i)})$, for example the first bit.

We recap in more detail how to recover a key byte $k_{u,v}$.

1. The attacker makes a guess on the key byte $k_{u,v}$ (there are 256 possible values).
2. The attacker partitions the t random 128-bit messages,

$$m_1 = (s_{u,v}^{(1)})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}}, \ldots, m_t = (s_{u,v}^{(t)})_{\substack{0 \le u \le 3 \\ 0 \le v \le 3}},$$

   into two sets, $\mathscr{S}_0$ and $\mathscr{S}_1$, given by

$$\mathscr{S}_0 = \left\{ m_i \mid \mathrm{bit}_1\left(\mathrm{S_{RD}}(s_{u,v}^{(i)})\right) = 0 \right\} \text{ and } \mathscr{S}_1 = \left\{ m_i \mid \mathrm{bit}_1\left(\mathrm{S_{RD}}(s_{u,v}^{(i)})\right) = 1 \right\}$$

   where $\mathrm{bit}_1\left(\mathrm{S_{RD}}(s_{u,v}^{(i)})\right)$ denotes the first bit of byte $\mathrm{S_{RD}}(s_{u,v}^{(i)})$.
3. Next, she evaluates the corresponding DPA trace

$$\Delta_P(t) := \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_1} - \langle \mathscr{C}_P(t) \rangle_{\mathscr{S}_0},$$

   and observes whether there are (significant) DPA peaks.
4. If not, the attacker goes back to step 1 with another guess for $k_{u,v}$. Otherwise, the attacker iterates the same attack to find the other key bytes.

The attack requires to evaluate at most 256 DPA traces to recover one key byte and so at most $256 \cdot 16 = 4096$ DPA traces to recover the cipher key.

13.4.3 Attack on an RSA Signature Scheme (2)

The attack we described in Section 13.2.2 readily applies by considering the power consumption as a side channel. We assume the Hamming-weight model. The two sets of messages can for example be defined as

$$\mathscr{S}_0 = \big\{m_i \mid \mathrm{lsb}(X_i) = 0\big\} \quad \text{and} \quad \mathscr{S}_1 = \big\{m_i \mid \mathrm{lsb}(X_i) = 1\big\},$$

where $X_i = \mu(m_i)^{(d_{k-1},\ldots,d_{k-n+1},1)_2} \bmod N$ and $\mathrm{lsb}(X_i)$ denote the least significant bit of X_i.

13.5 Countermeasures

Since the publication of side-channel attacks against cryptographic devices, numerous countermeasures have been devised. Countermeasures are available at both the hardware and the software levels but the sought-after goal is the same: to suppress the correlation between the side-channel information and the actual secret value being manipulated.

A first class of countermeasures consists in reducing the available side-channel signal. This implies that the implementations should behave regularly. In particular, branchings conditioned by secret values should be avoided.

A second class of countermeasures involves introducing noise. This includes power smoothing whose purpose is to render power traces smoother. The insertion of wait states (hardware level) or dummy cycles (software level) is another example. This technique is useful to make statistical attacks harder to implement as the signals should first be "re-aligned" (i.e., the introduced delays should be removed). One can also make use of an unstable clock to desynchronize the signals. At the software level, data masking is another popular countermeasure. For example, an RSA signature (cf. Section 13.2.2) can equivalently be evaluated as

$$S = \big[(\mu(m) + r_1 N)^{d + r_2 \phi(N)} \bmod (r_3 N)\big] \bmod N,$$

for three random values r_1, r_2 and r_3.

Efficiency is not the only criterion to deal with when developing cryptographic products. The implementations must also be resistant against side-channel attacks. To protect against the vast majority of known attacks, they must combine countermeasures of both classes described above. Moreover, experience shows that the best results are attained by mixing hardware and software protections.

Acknowledgments I would like to express my gratitude to my former colleagues of Gemplus and in particular to David Naccache, Francis Olivier and Michael Tunstall. I am also grateful to my colleagues of the Security Labs of Thomson for comments. The figures of this chapter are courtesy of Gemalto.

13.6 Exercises

1. Suppose that in the password verification routine of Algorithm 4, the comparison is done in a random order. More specifically, let $\mathfrak{S}$ be the set of permutations on the set $\{0,1,\ldots,7\}$. At each execution a permutation s is randomly chosen in $\mathfrak{S}$ and the comparison (cf. Line 2 of Algorithm 4) is replaced with
   2: **if** $(\widetilde{P}[s(j)] \neq P[s(j)])$ **then return** "`false`" .

   a. Do you think that this implementation is secure against timing attacks? Can you break it?
   b. Can you extend your attack if a random delay is added before returning the status?

2. Consider the timing attack given in Section 13.2.2 against an implementation of the "hash-and-sign" RSA signature scheme (Algorithm 5).

   a. The attack recovers one bit of secret exponent d at a time. Can you modify it to recover more than one bit at a time?
   b. Let $N = pq$ be a k-bit RSA modulus ($k \geq 1024$) where p and q are two (different) balanced secret primes satisfying $\gcd(e,(p-1)(q-1)) = 1$.
      i. Prove that for public exponent $e = 3$, the corresponding private exponent is given by $d := 3^{-1} \bmod \phi(N) = \frac{1+2(p-1)(q-1)}{3}$. Next, letting $\hat{d} = \frac{1+2N}{3}$, prove that $\hat{d} - d < 2^{\lfloor k/2 \rfloor + 1}$.
      ii. Explain how the previous relation can be used to speed up the attack when public exponent $e = 3$.
   c. The so-called square-and-multiply-*always* algorithm is a balanced version of Algorithm 5. It requires an additional register (for convenience, we write it as R_{-1}). It is obtained by replacing Lines 1 and 4 of Algorithm 5 with
      1: $R_0 \leftarrow 1; R_1 \leftarrow \mu(m); R_{-1} \leftarrow \mu(m)$
      4: $b \leftarrow d_j - 1; R_b \leftarrow R_b \cdot R_1 \pmod N$
      respectively. Can you mount a timing attack against this modified implementation?

3. Figure 13.1 represents four power traces of $f(x,0)$ for $x = 0$, 1, 7 and 255. You can observe that the highest consumption level is not always obtained for the same value of x (look at the different trace signatures). Does this give you indications on the leakage model?
4. The power trace of Figure 13.3 corresponds to an RSA exponentiation with private exponent $d = \texttt{2EC6915BF94A}\ldots$

a. How can you figure out the number of leading 0-bits? Namely, how can you determine that d starts with $\underline{00}101110\ldots = \texttt{2E}\ldots$ and not for example with $\underline{0}1011101\ldots = \texttt{5D}\ldots$?
b. Let ϕ denote Euler's totient function. Adding to d a random multiple of $\phi(N)$—or of $(ed-1)$ if $\phi(N)$ is not available—does not modify the result of an RSA exponentiation. Do you think it helps if, at each execution, private exponent d is randomized as $d \leftarrow d + r\phi(N)$ for a random integer r?

5. a. Explain why the circular shifting is done to the right when DES is used in decryption whereas it is done to the left for encryption. Why is there no circular shifting for the first round of DES^{-1} (see Table 13.1)?
   b. The DES key shifting presented in Algorithm 6 is susceptible to SPA. Propose an SPA-resistant implementation of it. Does your implementation resist DPA?
6. The bit-tracing method described in Section 13.4.1 is used to locate where in the power trace the ciphertext is transferred to RAM memory.
   a. The analysis requires the knowledge of the ciphertext. It is worth remarking that the ciphertext is known but not chosen. What is the impact on the analysis if the encryption algorithm is weak?
   b. As presented, the analysis considers a given bit of the first byte of the ciphertext to locate the beginning of the memory transfer. Propose another selection function so as to obtain higher peak values.
7. a. The DPA attack against AES in Section 13.4.2 assumes that the attacker has access to the input plaintexts. Can you adapt the attack in the case where the attacker only knows the output ciphertexts?
   b. It might be the case that different key byte candidates give rise to DPA peaks. In such a case, how could you identify the right candidate?
   c. Semantic security implies that encryption should be probabilistic. This can be achieved with AES by encrypting a 128-bit plaintext m under key K as $C = (c_1, c_2)$ with $c_1 = \text{AES}_K(r)$ and $c_2 = m \oplus r$ for a 128-bit random r. Plaintext m can then be recovered from ciphertext $C = (c_1, c_2)$ using key K as $m = c_2 \oplus \text{AES}_K^{-1}(c_1)$.
      i. Do you think that this implementation is secure against DPA attacks?
      ii. Can you mount a DPA attack against this randomized version of AES?
8. Give a detailed presentation of the DPA attack against the RSA signature scheme as described in Algorithm 5.

13.7 Projects

Power analysis requires specialized equipment which is not necessarily easily accessible. So, we will only consider timing information as a side channel. These projects can be done in C or using a higher-level language like Pari/GP [1].

1. a. Implement the password verification routine as presented in Algorithm 4.
   b. Mount the timing attack described in Section 13.2.1 against your implementation.
   c. Design a password verification routine that resists timing attacks.
2. a. Implement the Montgomery modular multiplication (see [10] or [9, Section 14.3.2] for details).
   b. Implement the RSA signature scheme as presented in Algorithm 5 using your implementation of the Montgomery modular multiplication. To simplify the implementation, you may assume that input messages are in $[0, N-1]$ and replace hash function μ with the identity map.
   c. Mount a timing attack against your implementation.
   d. Optimize your attack for the case $e = 3$ (see Problem 2(b)).
   e. Consider other exponentiation algorithms (e.g., [9, Section 14.6]) and mount timing attacks against the resulting implementations.

References

1. C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. Pari/GP. Freely available at URL http://pari.math.u-bordeaux.fr
2. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security*, pp. 62–73. ACM Press, 1993.
3. M. Bellare and P. Rogaway. The exact security of digital signatures. In *Advances in Cryptology – EUROCRYPT '96*, LNCS vol. 1070, pp. 399–416. Springer, 1996.
4. D. Boneh, G. Durfee, and Y. Frankel. Exposing an RSA private key given a small fraction of its bits. In K. Ohta and D. Pei, editors, *Advances in Cryptology – ASIACRYPT '98*, LNCS vol. 1514, pp. 25–34. Springer-Verlag, 1998.
5. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A practical implementation of the timing attack. In J. J. Quisquater and B. Schneier, editors, *Smart Card Research and Applications (CARDIS '98)*, LNCS, vol. 1820, pp. 167–182. Springer-Verlag, 2000.
6. G. Hachez and J.-J. Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000*, LNCS, vol. 1965, pp. 293–301. Springer-Verlag, 2000.
7. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, LNCS, vol. 1666, pp. 388–397. Springer-Verlag, 1999.
8. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, LNCS, vol. 1109, pp. 104–113. Springer-Verlag, 1996.

9. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. Online version available at URL http://www.cacr.math.uwaterloo.ca/hac/
10. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
11. National Bureau of Standards. Data encryption standard. Federal Information Processing Standards Publication 46, U.S. Department of Commerce, January 1977.
12. National Institute of Standards and Technology. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce, November 2001.
13. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
14. C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, 1999.

Chapter 14
Improved Techniques for Side-Channel Analysis

Pankaj Rohatgi

14.1 Introduction

Over the last several years, side-channel analysis has emerged as a major threat to securing sensitive information in hardware and systems. The list of side-channels that have been (re)discovered include timing [8] micro-architectural anomalies [1, 5, 12, 13], power consumption [9], electromagnetic emanations [2, 7, 14], optical [10, 11] and acoustic leakage [4]. These side-channels have been used to break implementations of all major cryptographic algorithms (such as DES, AES, RSA, Diffie-Hellman, Elliptic curves, COMP128, etc.) both in software and in hardware as well as for extracting information directly from peripherals. Concurrently a variety of side-channel analysis techniques have been developed to perform these attacks. These techniques include simple power/EM analysis (SPA/SEMA), differential power/EM analysis (DPA/DEMA), higher-order DPA/DEMA, inferential power analysis (IPA), partitioning attacks, collision attacks, hidden Markov model, etc.

In fact, side-channel analysis is so powerful that most attacks succeed, in practice, using only a fraction of the information present within the side-channel(s)! Typically, these techniques do not analyze the characteristics of the noise present within the side-channel signals, but try to remove it by averaging over a large number of samples. Related leakages that occur at different times in a side-channel trace are not combined to extract more information, and leakages from multiple side-channels are rarely combined. Therefore, if such techniques fail to break an implementation using a small number of side-channel signals, it cannot be assumed that the implementation is immune to side-channel attacks involving a limited number of side-channel traces. This question is particularly important to vendors, since there are several system-level side-channel countermeasures [9] based on nonlinear key updates that rely on the assumption that an adversary cannot extract the key from a single (or few) side-channel trace(s). This question is also pertinent to

IBM T. J. Watson Research Center
e-mail: rohatgi@us.ibm.com

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_14,

© Springer Science+Business Media, LLC 2009

implementations of stream ciphers such as RC4 that have a rapidly (and nonlinearly) evolving internal secret state, where a side-channel attack must be able to recover the state before it gets changed.

Answering such questions related to the fundamental capabilities and limits of side-channel attacks requires a deeper understanding of side-channel leakages from a device and an information-theoretic analysis of the optimal side-channel attacks that are possible against it. In this chapter, we describe the theoretical foundations for such an analysis by presenting a leakage model for CMOS devices and the maximum likelihood principle as the information theoretic basis for determining the optimal attacks and limits of side-channel analysis. We introduce the multivariate Gaussian noise assumption that makes it practical to apply the maximum likelihood principle to side-channel analysis. We then describe several applications of this approach. The first application, *template attacks*, shows how implementations of stream ciphers such as RC4 that are immune to simple and differential side-channel attacks could be broken using a single side-channel trace. Since this classical template attack has several practical shortcomings we also describe single-bit template attacks that may be suboptimal but much more practical. We then describe other applications of the maximum likelihood approach, such as an improved metric for DPA/DEMA attacks, the design and analysis of attacks involving multiple side-channels, and for information leakage assessment.

14.2 CMOS Devices: Side-Channel Leakage Perspective

Side-channels such as power and EM from a CMOS device are directly attributable to the currents flowing within the device as it operates. The two basic types of current flows include *intentional* flows, which are currents that flow in accordance to the circuit design as it performs the computation, and *leakage* currents, which are a property of the technology used to fabricate the device. In addition, there is nonlinear electromagnetic coupling between the different currents flowing within the device which causes amplitude and angle modulation that in turn gives rise to several EM side-channels. In addition there are variations within the different current flows due to thermal noise.

14.2.1 Intentional Current Flows

In CMOS devices, all data processing is typically controlled by a "square-wave"-shaped clock. From a logical perspective, each clock edge causes the device to perform an *elementary operation* resulting in a change in the state of the device. From a physical perspective, the clock edge triggers a state-dependent sequence of switching events that result in current flows within the device. These events are transient and a steady state is achieved before the next clock edge. At any clock cycle, the

events and resulting currents are dependent on only a small number of bits of the logic state of the device and not its entire state. These bits are termed *relevant bits* and consist of the bits of the state that change as well as the bits that influence the bits of the state that get changed. The set of *relevant bits* during a clock cycle constitute the *relevant state* of the device at that clock cycle.

14.2.2 Leakage Current Flows

In an ideal CMOS device, currents only flow when there is switching activity. However, due to shrinking feature sizes and usage of stressed silicon, there is a significant amount of current due to *leakage* even within the *inactive* parts of the circuit. The net leakage current within the circuit depends purely on the technology used and the size of the circuit. For our purposes we can approximate leakage current within a CMOS device as a constant plus a small Gaussian noise term that is uncorrelated to the activity occurring within the active part of the circuit.

14.2.3 Information Leakage in Power and EM Side-Channels

The power side-channel can be viewed as an aggregate measure of all the currents flowing within the device. Of these currents, only the intentional currents can provide information about the *relevant state* of the device. However, due to aggregation of currents and noise and due to impedances within the circuit and power grid, the influence of weak individual intentional currents on the power side-channel can be quite small. For understanding information leakage from EM, one only needs to consider coupling effects that involve at least one *intentional* current. Even a single EM sensor can pick up multiple and distinct mixtures of coupling effects over the entire EM spectrum.

As a very good first approximation, both power and EM side-channel emanations during a clock cycle carry information *only* about the relevant state of the device during the clock cycle and not the other parts of the device state.

This is strongly supported by the experimental results which show that algorithmic bits are significantly correlated to the power/EM signals *only* during the clock cycles where the bits are actively involved in a computation. While an algorithmic bit may leak to different degrees in different power and EM channels at different parts of the computation, it never leaks when it is inactive. For example, Figure 14.1 shows a power trace (in gray) overlaid with the contribution of a particular bit to power trace (in black). This bit only impacts the power trace during some clock cycles where it is part of the relevant state and not during all clock cycles. Figure 14.2 which plots the leakage of the same bit in different power/EM side-channels shows that the extent of leakage is different for different side-channels.

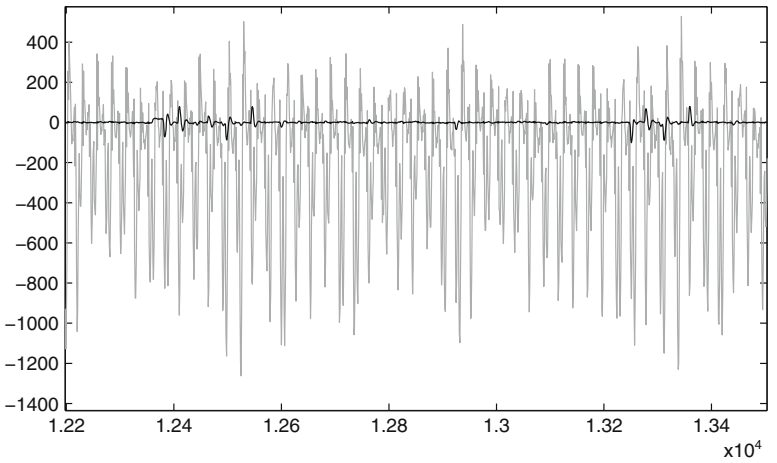

Fig. 14.1 Power side-channel (gray) overlayed with contribution from single relevant bit (black).

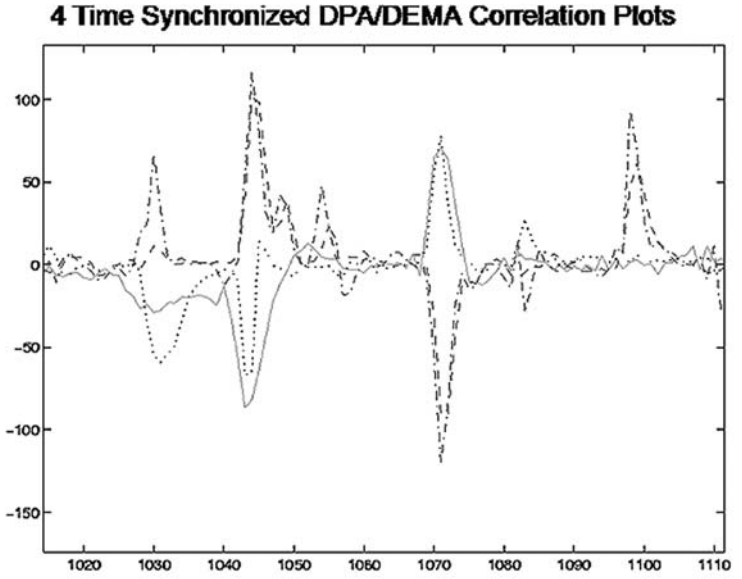

Fig. 14.2 DPA and three DEMA correlation curves (aligned).

14.3 Characterizing Side-Channel Leakage Using Maximum Likelihood

14.3.1 Adversarial Model

Given the side-channel leakage model above, it becomes natural to formulate side-channel attacks in terms of how successful an adversary can be in obtaining information about the *relevant state* using side-channels. For example, an adversary may be interested in the LSB of the data bus during a LOAD instruction or he may be interested in finding out the address of the data being loaded.

In general, the adversary would like to use side-channels to extract information about the relevant state of a device when it is performing an elementary operation given some prior knowledge about the relevant state. This is a classical inferencing problem, but for simplicity, we can assume that the adversary is attempting to find information about parts of the relevant state that are completely unknown. In this case the problem is naturally formulated as a hypothesis testing problem as follows:

The adversarial model consists of two phases. The first phase, known as the *profiling phase*, is a training phase for the adversary. He is given a *training device* identical to a *target device*, an elementary operation, k distinct probability distributions $B_1, \ldots, B_k$ on the relevant states from which the elementary operation can be invoked and a set of sensors for monitoring side-channel signals.

The adversary can invoke the elementary operation, on the training device, starting from any relevant state. It is expected that adversary uses this phase to prepare an attack.

In the second phase, known as *the hypothesis testing phase*, the adversary is given a *target device* and the same set of sensors. He is allowed to make a *bounded number L* of invocations to the same elementary operation on the target device starting from a relevant state that is drawn *independently* for each invocation according to exactly one of the k distributions $B_1, \ldots, B_k$. The choice of distribution is completely unknown to the adversary (i.e., a priori, each distribution is equally likely to be chosen with probability $\frac{1}{k}$) and his task is to use the signals on the sensors to select the correct hypothesis $(H_1, \ldots, H_k)$ for the distribution being $B_1, \ldots, B_k$. The utility of the side-channels to extract this information can then be measured in terms of the success probability achieved by the adversary as a function of the number of invocations L.

14.3.2 Maximum Likelihood and Best Attack Strategy

Assume that an adversary acquires L statistically independent sets of sensor signals $\mathbf{O}_i, i = 1, \ldots, L$. These L sets of signals may correspond to L invocations of an operation on the target device. Also assume that there are K equally likely hypotheses H_k, $k = 1, \ldots, K$, on the origin of these signals. Let $p(\mathbf{O}|H)$ be the probability

distribution of the sensor signals under the hypothesis H. Under these assumptions, the *maximum likelihood hypothesis* test [15] is optimal and it decides in favor of the hypothesis H_k, if for all j, where $1 \leq j \leq K$

$$\prod_{i=1}^{L} p(\mathbf{O}_i|H_k) \geq \prod_{i=1}^{L} p(\mathbf{O}_i|H_j) \tag{14.1}$$

i.e., H_k is the hypothesis under which the actual observations have the highest probability of occurring.

While the maximum likelihood test is optimal, it is usually impractical as an exact characterization of the probability distribution of the sensor signals $\mathbf{O}$ may be infeasible. Such a characterization has to capture the nature of each of the sensor signals and the dependencies among them. This could further be complicated by the fact that, in addition to the thermal noise, the sensor signals could also display additional structure due to the interplay between properties of the device and those of the distributions of the relevant states.

It turns out that in practice one can obtain near-optimal results by making the right assumptions about the sensor signals. Such assumptions greatly simplify the task of hypothesis testing by requiring only a partial characterization of sensor signals.

14.3.3 Gaussian Assumption

One such widely applicable assumption is the *Gaussian assumption* which states that under the hypothesis H, the sensor signal $\mathbf{O}$ has a multivariate Gaussian distribution with mean μ_H and a covariance matrix Σ_H [15]. A multivariate Gaussian distribution $p(\cdot|H)$ has the following form:

$$p(\mathbf{o}|H) = \frac{1}{\sqrt{(2\pi)^n|\Sigma_H|}} \exp(-\frac{1}{2}(\mathbf{o}-\mu_H)^T \Sigma_H^{-1}(\mathbf{o}-\mu_H)), \quad \mathbf{o} \in \mathscr{R}^n \tag{14.2}$$

where $|\Sigma_H|$ denotes the determinant of Σ_H and Σ_H^{-1} denotes the inverse of Σ_H.

The Gaussian assumption holds for a large number of devices and hypotheses encountered in the practice. It can be shown that under the Gaussian assumption, the maximum likelihood hypothesis testing for a single observation $\mathbf{O}$ and two equally likely hypothesis H_0 and H_1[1] simplifies to the following comparison:

$$(\mathbf{O}-\mu_{H_0})^T \Sigma_{H_0}^{-1}(\mathbf{O}-\mu_{H_0}) - (\mathbf{O}-\mu_{H_1})^T \Sigma_{H_1}^{-1}(\mathbf{O}-\mu_{H_1}) \geq \ln(|\Sigma_{H_1}|) - \ln(|\Sigma_{H_0}|) \tag{14.3}$$

where a decision is made in favor of H_1 if the above comparison is true, and in favor of H_0 otherwise.

[1] Generalizations to multiple observations and more than two hypotheses are straightforward.

In signal processing it is common to treat the observed trace $\mathbf{O}$ as consisting of a mean signal component that depends purely on the operation being performed on the device and is fixed across multiple invocations and a noise component which can differ on each invocation. Noise is best modeled as a random sample drawn from a noise probability distribution having a mean of zero. In the equations above, if hypothesis H was correct then the mean signal component would be μ_H and the noise component in each sample, i.e., $\mathbf{O} - \mu_H$, would also have a multivariate Gaussian distribution with mean 0 and the same covariance matrix Σ_H. Thus the Gaussian assumption made here in the context of the signal characterization is often alternatively referred to as the Gaussian noise assumption.

In many cases of practical interest, noise in the sensor signals does not depend on the hypothesis, that is, $\Sigma_{H_0} = \Sigma_{H_1} = \Sigma_N$. In such cases, the following well-known result from statistics gives the probability of error in maximum likelihood hypothesis testing [15]:

Fact 1 *For equally likely binary hypotheses, the probability of error in the maximum likelihood testing is given by*

$$P_\varepsilon = \frac{1}{2}\operatorname{erfc}\left(\frac{\Delta}{2\sqrt{2}}\right) \tag{14.4}$$

where $\Delta^2 = (\mu_{H_1} - \mu_{H_0})^T \Sigma_N^{-1} (\mu_{H_1} - \mu_{H_0})$ *and* $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$*, where*

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

is the error function. Note that Δ^2 *has a nice interpretation as the optimal signal-to-noise ratio that an adversary can achieve under the Gaussian assumption.*

In the rest of this chapter we will describe multiple applications of this characterization of side-channels and their leakage.

14.4 Template Attacks

The motivation behind the development of template attacks was that in several instances only a single (or a few) side-channel sample is available for carrying out an attack against a device. This situation arises naturally in the case of stream ciphers where the internal secret state keeps changing as the key stream is generated and in protocols where ephemeral keys are used. In addition, there are some system-level countermeasures that try to limit side-channel exposure by limiting the use of a particular key [9]. In such cases, the implementations can be easily made to be secure against traditional simple/differential attacks since typically the differences between signal levels with different keys/data is usually lower than the level of noise. This noise cannot be eliminated by averaging since there is only one or a very limited number of traces available.

For attacking such implementations we convert the model described in the last section into an attack technique called template attacks. If an adversary had infinite resources, the template attack would work using this basic principle: Suppose there is a crypto implementation on many "identical" devices and the adversary has access to one such device on which he can perform experiments and he is also given a single (or few) side-channel sample(s) S from a target device with an unknown key. The adversary uses the test device to build signal/noise models or *templates* for side-channel signals produced by the test device for *all* possible values of the key and uses the maximum likelihood to determine which key is used in the target sample.

Clearly, since key sizes are large, it is infeasible to build templates for all possible keys. Practical template attacks have to meld this basic attack principle with the details of the cryptographic algorithm being attacked. Typically this is done in an iterative fashion, where at each stage, the adversary starts with a small candidate set of prefixes for the key and ends with another small candidate set of larger-sized prefixes of the key. At the end of this process, the adversary has a limited set of complete keys that he can exhaustively test. In the beginning the candidate set is empty. At each step, the adversary uses the test device to identify a small sub-section of the sample S that depends only on a few unknown key bits. By experimenting with the test device, he builds signal templates corresponding to his set of candidate key prefixes extended by all possible value of the unknown key bits. The templates consist of the mean signal and (multivariate Gaussian) noise probability distributions for each of these extended prefixes of the key. He then compares these templates with the signal S and uses the maximum likelihood principle to retain only a small set of those prefixes that match S the best. Thus template attacks essentially use an extend-and-prune strategy directed by the single sample S to be attacked: the adversary extends candidate key prefixes by all possible values of a limited number of unknown key bits, builds templates, and uses template classification to prune the set of choices for these larger key prefixes. The success of this approach depends on the effectiveness of the pruning strategy in controlling the combinatorial explosion that occurs during the extension process.

Template attacks are particularly effective on implementations of cryptographic algorithms due to their *contamination* and *diffusion* properties. Contamination refers to key-dependent leakages which can be observed over multiple cycles in a section of computation. Additionally, other variables affected by the key, such as key-dependent table indices and values, cause further contamination at other cycles. The extent of contamination controls the success of the pruning of the fresh key bits introduced in the expansion phase. However, it is to be expected that if two keys are almost the same, that even with the effects of contamination, pruning at this stage may not be able to eliminate one of them. Diffusion is the well-known cryptographic property wherein small differences in key bits are increasingly magnified in subsequent portions of the computation. Even if certain candidates for key prefixes were not eliminated by contamination effects, diffusion will ensure that the wrong key prefixes get pruned rapidly at later stages.

We now provide the details on how such attacks could be carried out in practice by means of an example.

14.4.1 Classical Template Attacks: The Case of RC4

Consider an implementation of the stream cipher RC4. While there are cryptanalytic results on RC4 based on minor statistical weaknesses, none of these are useful for side-channel attacks. A well-designed system, RC4 implementations are also quite easy to secure against SPA- and DPA-style attacks. This is because initializing the 256-byte internal state of RC4 with a secret key is simple enough to be implemented using low leakage instructions in a key-independent manner. This makes SPA unlikely. After initialization, the only secret is the internal state. However, this secret state evolves very rapidly as the cipher outputs more bytes. This rapidly evolving secret state is outside the control of the adversary. This provides inherent immunity against statistical attacks such as DPA, since the adversary cannot freeze the active part of the state to collect multiple samples to eliminate the noise. For RC4, the best that an adversary can hope for is to obtain a *single* sample of the side-channel leakage during the key initialization phase and attempt to recover the key from that single sample.

We now describe how template attacks apply against RC4's state initialization routine. RC4 operates on a 256-byte state table T to generate a pseudo-random stream of bytes that is then XORed with the plaintext. Table T is initially fixed, and in the state initialization routine, a variable length key (1 to 256 bytes) is used to update T using the pseudo code below:

```
i1 = 0
i2 = 0
for ctr = 0 to 255
  i2 = (key[i1] + T[ctr] + i2) mod 256
  swap_byte(T[ctr], T[i2]);
  i1 = (i1 + 1) mod (key_data_len)
endfor
```

A portion of the corresponding power side-channel signal (plotted in gray) and the sample noise (plotted in black) for the first six iterations of the loop is shown in Figure 14.3.

First it needs to be verified that simple side-channel analysis techniques will not work on this implementation. This can be easily seen in Figure 14.4 which plots the noise level for the first six iterations in a power sample in gray and plots in black the difference between the signals for two different keys A and B that differ only in the first byte. The figure clearly shows that the level of noise in the first iteration (time 0 to 20 µs) far exceeds the differences between the signals for keys A and B in that iteration; so SPA will not have been able to determine which key byte was used in the first iteration. In fact, in [6], it was stated that averages of several tens of samples would be needed to reduce the level of noise below the signal differences.

RC4 is, however, an ideal candidate for a signal classification-based attack. Notice from the code snippet above that the key byte used in each iteration influences the computation (and is part of the relevant state) multiple times within a loop. For example, loading of the key byte, the computation of index `i2`, and the use of `i2`

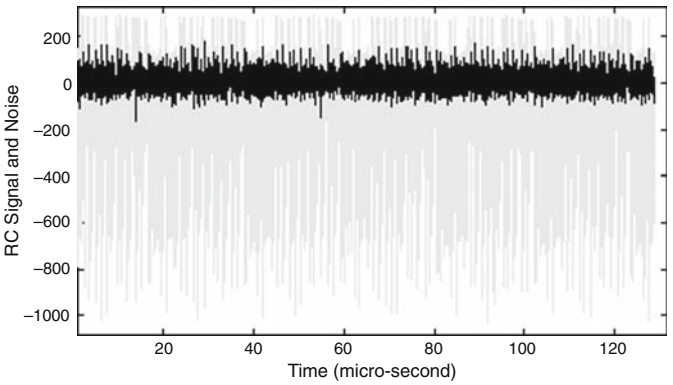

Fig. 14.3 Power signal (gray) and noise (black) during first six iterations of RC4 state initialization loop.

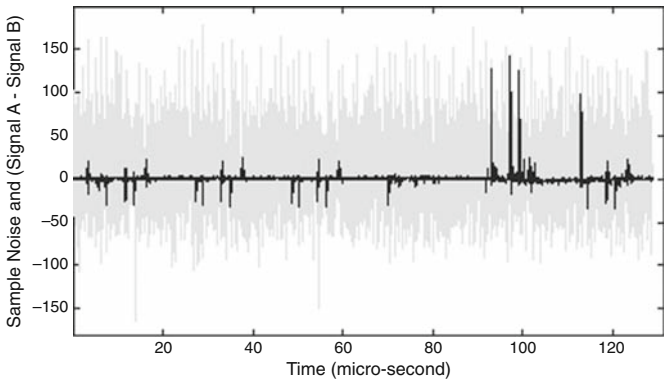

Fig. 14.4 Sample noise (gray) vs. signal differences (black) between keys A and B in first six rounds.

in swapping the bytes of the state table T all contaminate the side-channel at different cycles in the iteration. Thus RC4 demonstrates good *contamination* for the individual key bytes. Further, the use of `i2` and the state in subsequent iterations, and the fact that RC4 is a well-designed stream cipher, quickly propagates small key differences to cause diffusion. This analysis is borne out in practice as is shown in Figure 14.5 which plots the signal for the first six iterations for the key A in gray and the difference between the signals for key A and key B in black. Keys A and B differ only in the first byte and a small difference signal is clearly visible in the figure in the first iteration (0 to 20 μs). The important point to note is that even though the magnitude of the difference signal is small in the first iteration, significant differences appear at many different places in the first iteration, which indicates good contamination. The next point to note is that by the time the fifth iteration is reached, the difference signal has become quite large, indicating good diffusion.

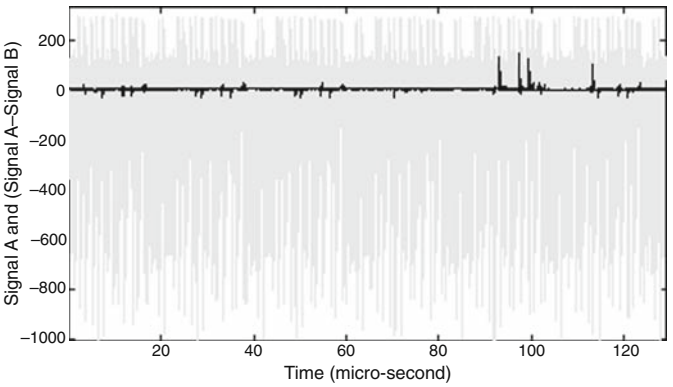

Fig. 14.5 Signal for key A (gray) vs. signal differences (black) between keys A and B in first six rounds.

The template attacks on this RC4 implementation works by building templates for the signal and noise for around 42 sample points in each iteration of RC4 state initialization routine that takes an input a fresh (unknown) key byte. These are the points where significant differences arise for different keys as shown in Figure 14.5.

A first attempt to use a statistical model where the noise at these 42 points was treated independently (i.e., Σ_H is a diagonal variance matrix) to classify the (unknown) key byte yielded poor results as shown in Table 14.1. The table here shows the classification rates assuming there were only five possible values of the key byte. Here the classification errors were as much as 35% for pairs of keys with few bit differences. Somewhat more encouraging was the fact that even this limited statistical model was fairly good (100% successful) at distinguishing between key byes that were very different.

Next the full multivariate noise approach was applied. In the experiment, there were 10 choices for the first key byte, as shown in column 1 of Table 14.2. They are carefully chosen to be very close and yielded poor results with the univariate statistics. For each key byte, 2000 side-channel signals were collected and analyzed at the same 42 points in time. The mean of the 2000 samples was used as the average signal for that key (μ_H) and the covariance matrix (Σ_H) was also computed from these signals. To obtain statistics on how well this approach works, the templates were used to classify tens of thousands of samples drawn using one of the

Table 14.1 Classification probability of five competing hypotheses using univariate statistics. Entry (i,j) is probability of classifying samples with key i as one with key j.

Key byte	1111 1110	1110 1110	1101 1110	1011 1110	0001 0000
1111 1110	0.86	0.04	0.07	0.03	0.00
1110 1110	0.06	0.65	0.10	0.20	0.00
1101 1110	0.08	0.16	0.68	0.09	0.00
1011 1110	0.10	0.11	0.08	0.71	0.00
0001 0000	0.00	0.00	0.00	0.00	1.00

Table 14.2 Percentage of samples for which the correct hypothesis is retained under different ball sizes with 10 competing hypotheses.

Key byte	Ball size $c = 1$	Ball size $c = e^6$	Ball size $c = e^{12}$	Ball size $c = e^{24}$
1111 1110	98.62	99.46	99.88	99.94
1110 1110	98.34	99.82	99.88	99.88
1101 1110	99.16	100.00	100.00	100.00
1011 1110	98.14	99.52	99.82	100.00
0111 1110	99.58	99.76	99.89	99.94
1111 1101	99.70	99.94	99.94	99.94
1111 1011	99.64	99.82	99.82	99.89
1111 0111	100.00	100.00	100.00	100.00
1110 1101	99.76	99.82	99.88	99.88
1110 1011	99.94	100.00	100.00	100.00
Average	99.29	99.81	99.91	99.95

10 choices as the first key byte. Column 2 in Table 14.2 summarizes the results of the classification experiments for this set of 10 key choices. Since the values were carefully chosen to reflect the worst case, these results can be extrapolated to the case of 256 different values of the key byte. Column 2 in Table 14.4 is an extrapolation of our results for the case of 256 different templates by making pessimistic assumptions about the number of "close" keys. In practice the actual results should be much better.

To iteratively apply the approach a first heuristic would be to retain only the most likely hypothesis, i.e., with highest likelihood probability. Even with such a drastic pruning approach, average classification success probability is 99.3% with these 10 hypotheses and worst-case probability was 98.1%. Detailed results are described in column 2 of Table 14.2. One gets reasonable results even if we use this extreme pruning strategy in each iteration of the extend-and-prune approach. Extrapolating, as shown in column 2 of Table 14.4, one can expect the average error probability of the closest hypothesis approach to be about 5–6% when we consider all 256 possible values, since pessimistically one expects around 50–60 keys to be "close" to any key. By bounding the error probability over many iterations by the sum of error in each iteration, it can be seen that when the number of key bytes is small this can be used to extract all key bytes. For example, one can do better than 50% for about eight bytes of key material.

With a little more effort, much better results can be obtained by using a *ball approach* to pruning. In this approach, a constant c is chosen and if the best hypotheses has probability P then all hypotheses that have probability P/c are retained. This approach is analogous to retaining all hypotheses which are a certain radius away from the top hypothesis and hence the term *ball approach*. The columns 2, 3, 4, and 5 of Table 14.2 showing success probability of retaining the correct hypothesis for balls with different values of c, with column 2 corresponding to $c = 1$ and retaining only the most likely hypothesis. When $c = e^6$, the average success probability has improved to better than 99.8% with the worst-case probability being 99.5%. As shown in Table 14.3 the average number of hypotheses that we retain is still close to 1

Table 14.3 Expected number of hypotheses retained under different ball sizes for 10 competing hypothesis.

Ball size $c = 1$	Ball size $c = e^6$	Ball size $c = e^{12}$	Ball Size $c = e^{24}$
1	1.041	1.158	1.842

Table 14.4 Extrapolated results for 256 competing hypotheses.

	Ball size $c = 1$	Ball size $c = e^6$	Ball size $c = e^{12}$	Ball size $c = e^{24}$
Success prob.	95.02	98.67	99.37	99.65
Retained hypotheses	1	1.29	2.11	6.89

for balls of size e^6 and e^{12}. Again, using an estimate of about $50 - 60$ close keys, we can extrapolate these results as done in Table 14.4. For example, choosing the ball size e^6, with good probability we expect to retain at most 1.5 hypotheses on the average, yet we are guaranteed to retain the correct hypothesis with probability at least 98.67%. Using this approach independently in each iteration, we can correctly classify keys of size n bytes with expected probability around $(100–1.33n)\%$ and the number of remaining hypotheses would grow no more than $(1.5)^k$, which is substantially better than 2^{8k} (the entropy of the key).

14.4.2 Single-Bit Templates and Applications

The classical template attack described above suffers from several drawbacks. One major drawback is that the methodology requires an iterative approach that attempts to make test device's computation identical to that of the target device. This makes the attack tedious, iterative, and online. For example, in the RC4 case 256 templates for each unknown byte have to be constructed and templates for later bytes cannot be constructed until the earlier bytes have been attacked. Another drawback is that classical template attacks cannot handle randomized implementations, since the attacker cannot force a test device to produce the same randomness as the target device.

Single-bit template attacks are an attempt to get past these limitations at the cost of reduced classification accuracy. These attacks are based on an empirical observation that after a successful DPA attack on an algorithmic bit b, the DPA peaks themselves could be used to create binary templates to extract the bit b directly from any signal! This means that once a particular implementation/device has been attacked using DPA (say using a test card) one can predict the internal bits occurring within a single trace from an identical implementation/device.

We illustrate the attack by means of an example. Consider an unprotected implementation of DES on smartcard A. Consider the 32 s-box output bits of the DES computation in round one. For the unprotected DES implementation, one can easily perform DPA for each of the 32 output bits. Correspondingly, one can build a pair of templates for each output bit corresponding to the bit being equal to 0 and 1,

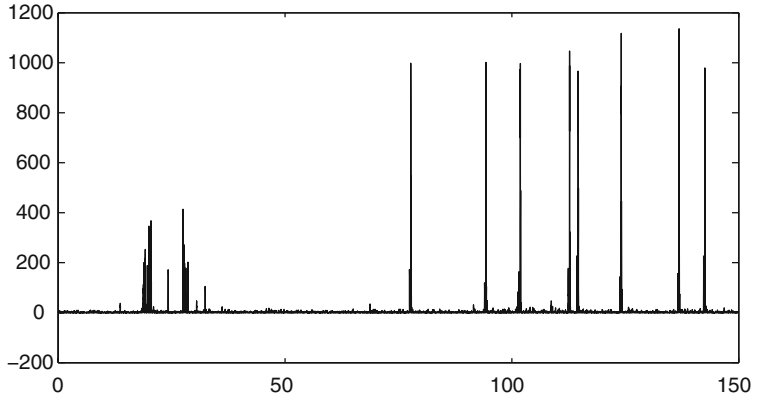

Fig. 14.6 Improved DPA metric of *s*-box 1, bit 0 of the test device. Time in µs.

respectively. In order to build these templates, a DPA attack was performed on each output bit using a DPA metric that we will be describing in Section 14.5.

Figure 14.6 displays the DPA metric of *s*-box 1, bit 0. The figure reveals several points in time that clearly correlate with the selected *s*-box output bit. In the experiments, the 50 highest peaks from this DPA metric were selected to be the points that were incorporated into the pair of templates (bit $= 0$ and bit $= 1$) for that bit. Templates were built for each *s*-box output bit using a single set of 1400 side-channel samples.

To estimate classification success rate, a set of 100 fresh random side-channel samples were collected from the same device and all 32 *s*-box output bits were classified using the templates developed earlier. The classification success rates $\eta_{S_i b_j}$ for the ith *s*-box and jth bit, $1 \leq i \leq 8$ and $0 \leq j \leq 3$, together with the corresponding entropy loss are shown in Table 14.5. The classification success rates ranged from 0.72 to 1.00; in the worst case *s*-box 3, bit 3 and *s*-box 6, bit 0 were predicted correctly for only 72 of the 100 samples. From these results, the probability that the entire 32-bit output of all *s*-boxes is classified correctly is $\prod_{i=1}^{8} \prod_{j=0}^{3} \eta_{S_i b_j} = 0.0154$ which although small is still 66 million times higher than a random guess.

These results can also be viewed in terms of entropy loss. For a particular bit, if the classification success rate is p, then its corresponding entropy loss is given by $1 + (1-p)\log_2(1-p) + p\log_2(p)$. To compute the entropy loss for multiple bits we

Table 14.5 *s*-Box output bit classification success rates and entropy loss.

	s-box 1	**s-box 2**	**s-box 3**	**s-box 4**	**s-box 5**	**s-box 6**	**s-box 7**	**s-box 8**
bit 0	1.00	0.91	0.88	0.93	0.77	0.72	0.80	0.84
bit 1	0.98	0.88	0.92	0.94	1.00	0.92	0.97	0.77
bit 2	0.75	0.89	0.99	0.92	0.95	0.83	0.90	0.79
bit 3	0.90	0.91	0.72	0.85	0.83	0.86	1.00	0.89
Entropy loss	2.57	2.10	2.13	2.30	2.28	1.50	2.61	1.35

can add the individual losses (this corresponds to the worst case where classification of different bits is independent). From this formula, we can see that 16.8 bits of entropy has been lost from the 48 bits of the DES key used in the first round (out of a maximum possible loss of 32 bits if the classification was perfect). The loss of entropy of the keyspace can be translated into reduced expected computational cost of a guided exhaustive search through the entire keyspace that examines more likely keys earlier than the less likely keys.

For DES implementations, the attack can be improved substantially. Templates can be built not just for round 1, *s*-box output bits but also for other bits such as the data bits fed to the second round. These templates will further narrow down the possibilities for the 48 key bits used in the first round. In addition, templates can be built for the corresponding DPA attacks on the last two rounds of DES (which utilize another 48-bit size subset of the key) and so on. Depending on the implementation, single-bit templates can also be built directly for the key bits that are likely to be highly effective since the same key bits show up in multiple locations in a round and across multiple rounds.

To summarize, single-bit template attacks are capable of classifying a single bit from a single side-channel sample with high probability even though the influence of a single bit on the side-channel signal at a point in time is very small and could be masked by several sources of noise including variation in adjacent bits. Cryptographic algorithms with high contamination properties such as DES are ideally suited for single-bit classification. Multiple precomputed single-bit templates can lead to practical guided keyspace search algorithms using only a single sample from the target device. Moreover, single-bit attacks when combined with other attacks can result in much more devastating attacks such as template-enhanced DPA [3] that use a DPA-like attack technique to overcome the random masking countermeasure, provided an adversary can acquire a single test card with a faulty RNG.

14.5 Improved DPA/DEMA Metric

In Section 14.4, when discussing template attacks, we assumed that the adversary had access to a test device identical to the target device and that he could carry out a profiling stage using the test device. In many circumstances, access to a test device may not be possible. In such cases, a DPA-style attack is preferred since it assumes no prior knowledge of device characteristics or implementation. In this section, we apply theory from Section 14.3 to optimize the analysis of existing single-channel DPA attacks.

14.5.1 Improving DPA

In the traditional DPA attack, an adversary collects a set of N signals, $\mathbf{O}_i, i = 1, \ldots, N$ emanating from a given channel. Assume that the signals are normalized to have zero sample average over all N signals. For each hypothesis H under consideration,

the N signals are divided into two bins, termed the 0-bin and the 1-bin, with $N_{H,0}$ and $N_{H,1}$ samples, respectively. Let $\mu_{H,0}[j]$ and $\mu_{H,1}[j]$ be the sample means of signals in the 0-bin and the 1-bin, respectively, for the hypothesis H. The next step in the DPA attack consists of computing the differences of sample means $\mu_H[j] = \mu_{H,0}[j] - \mu_{H,1}[j]$ for all hypotheses and deciding in favor of the hypothesis H_i if $|\mu_{H_i}[j]|$ has the largest peak among all differences of means. In other words, the decision metric for the hypothesis H at time j is given by

$$M_H[j] = \Big(\mu_{H,0}[j] - \mu_{H,1}[j]\Big)^2 \tag{14.5}$$

and the decision is made in favor of the hypothesis H_i if for some value of j, say j_0, $M_{H_i}[j_0] >= M_H[j]$ for all H and j.

The traditional DPA attack and its variations have been successfully applied to attack several cryptographic implementations. However, by using the theory developed in Section 14.3, the effectiveness of traditional DPA can be increased significantly.

Before proceeding further, assume a void hypothesis H_v which corresponds to a random bifurcation of the N signals into the 0-bin and the 1-bin. Using the Gaussian assumption and Equation (14.3), the metric of a hypothesis H_i with respect to the null hypothesis at time j is given by

$$M_{H_i}[j] = \frac{\Big(\mu_{H_i}[j] - E[\mu_{H_v}[j]]\Big)^2}{V[\mu_{H_v}[j]]} - \frac{\Big(\mu_{H_i}[j] - E[\mu_{H_i}[j]]\Big)^2}{V[\mu_{H_i}[j]]} - \ln\Big(\frac{V[\mu_{H_i}[j]]}{V[\mu_{H_v}[j]]}\Big) \tag{14.6}$$

In order to compute this metric, we need the values of the following parameters: $E[\mu_{H_v}[j]]$, $V[\mu_{H_v}[j]]$, $E[\mu_H[j]]$, and $V[\mu_H[j]]$. Since in the DPA attack, the adversary skips the profiling phase of the attack, Equation (14.6) is not directly applicable. In such cases, the theory suggests that unknown parameters of the test equation be estimated directly from the collected signals. If the adversary uses a maximum likelihood estimate of these parameters, then the resulting test is referred to as the generalized maximum likelihood testing.

For the DPA attack, calculating the maximum likelihood estimate of the test parameters involves solving a set of nonlinear coupled equations. Therefore, instead of using the maximum likelihood estimates of these parameters, we use sample estimates as follows: Let $\sigma^2_{H,0}[j]$ and $\sigma^2_{H,1}[j]$ be the sample variances of the signals in the 0-bin and the 1-bin, respectively, at time j for hypothesis H. We propose the following sample estimators[2] of parameters in Equation (14.6):

$$\begin{aligned} E[\mu_H[j]] &= \mu_H[j] \\ V[\mu_H[j]] &= \frac{\sigma^2_{H,0}[j]}{N_0} + \frac{\sigma^2_{H,1}[j]}{N_1} \end{aligned} \tag{14.7}$$

[2] We omit the derivation of these estimators as the derivation is tedious and follows from straightforward algebraic manipulations.

Substituting these in Equation (14.6), we get the following formula for the metric:

$$M_{H_i}[j] = \frac{\left(\mu_{H_i}[j] - \mu_{H_v}[j]\right)^2}{\frac{\sigma^2_{H_v,0}[j]}{N_0} + \frac{\sigma^2_{H_v,1}[j]}{N_1}} - \ln\left(\frac{\frac{\sigma^2_{H_i,0}[j]}{N_0} + \frac{\sigma^2_{H_i,1}[j]}{N_1}}{\frac{\sigma^2_{H_v,0}[j]}{N_0} + \frac{\sigma^2_{H_v,1}[j]}{N_1}}\right) \tag{14.8}$$

Table 14.6 shows the results of applying this method to attacking the s-box lookup for a DES implementation. The first column shows the bit being predicted, the second shows the number of samples required for the correct key hypothesis to emerge as the winner under the traditional DPA metric, while the third column shows the number of samples needed with the new metric. Clearly by using a better metric, our improvement in the DPA attack reduces the number of signals needed by a factor of 1.4–3.

Table 14.6 DPA results, mean difference vs. approximate generalized maximum likelihood.

S-box hyp.	Min samples (mean diff.)	Min samples(Max. Likl.)
S1,B3	640	350
S2,B3	630	210
S7,B3	110	40
S8,B3	130	90

14.6 Multi-Channel Attacks

As we have seen, there are several side-channels including power and multiple EM channels that carry somewhat different information. Given this multitude of information-bearing signals, a natural question to ask is how these multiple leakages could be combined to enable better attacks. In addition, since each additional sensor and side-channel signal used for analysis raises the cost and complexity of an attack, another important question is how a resource-limited adversary could best select the sensors and side-channels to mount an attack. In addition, system designers would like to know how much information could leak to an adversary who is able to place a set of side-channel sensors to capture information from the device. In this section we will use the theory developed in Section 14.3 to answer all these questions.

14.6.1 Multiple Channel Selection

Consider a resource-limited adversary who can select at most M channels for an attack. When viewed in terms of our model, this problem conceptually has a very simple solution: The adversary should choose those M channels that minimize his probability of error in the maximum likelihood testing.

This apparently simple technique can be quite subtle and tricky in practice. Clearly, in situations where a well-prepared adversary has nicely characterized and approximated signals from each of the channels under each hypothesis and the corresponding joint noise probability distribution between all the channels, the adversary can also calculate the error probability for each possible choice of M channels, at least for small M. For example, if the noise is Gaussian and independent of the hypothesis, then from Equation (14.4), since $\mathrm{erfc}(\cdot)$ decreases exponentially with Δ, the goal of an adversary limited to just two channels would be to choose channels in such a manner as to maximize the output signal-to-noise ratio Δ^2.

If instead of a rigorous approach, channels are selected by heuristic techniques, then the resulting selection could be sub-optimal for various subtle reasons. First, different side-channels could leak different aspects of information relative to the hypotheses being tested and sometimes there could be value in combining channels which provide widely dissimilar information rather than combining those which provide similar but partial information. Second, even if many channels provide the same information, picking multiple channels from this set could still be valuable since that may be almost as good as having the ability to make multiple invocations of the device with the same data and collecting a single side-channel. Even for the case where only two side-channels can be selected, the optimal choice is quite tricky and subtle as shown by the example below where the naive approach of choosing the two signals with best signal-to-noise ratios is shown to be sub-optimal.

Example 14.1. Consider the case where an adversary can collect two signals $[O_1, O_2]^T$ at a single point in time, such that under the hypothesis H_0, $O_k = N_k$ for $k = 1, 2$ and under the hypothesis H_1, $O_k = S_k + N_k$. Assume that $\mathbf{N_i} = (N_1, N_2)^T$ has zero mean multivariate Gaussian distribution with

$$\Sigma_N = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

Note that O_1 and O_2 have signal-to-noise ratios of S_1^2 and S_2^2, respectively. After some algebraic manipulations, we get

$$\Delta^2 = \frac{(S_1 + S_2)^2}{2(1+\rho)} + \frac{(S_1 - S_2)^2}{2(1-\rho)} \tag{14.9}$$

Now, consider the case of an adversary who discovers two AM-modulated carrier frequencies which are close and carry compromising information, both of which have very high and equally good signal-to-noise ratios ($S_1 = S_2$) and another AM-modulated carrier in a very different band with a lower signal-to-noise ratio. An intuitive approach would be to pick the two carriers with high signal-to-noise ratios. In this case $S_1 = S_2$ and we get $\Delta^2 = 2S_1^2/(1+\rho)$. Since both signals originate from carriers of similar frequencies, the noise that they carry will have a high correlation coefficient ρ, which reduces Δ^2 at the *output*. On the other hand, if the adversary collects one signal from a good carrier and the other from the worse quality carrier

in the different band, then the noise correlation is likely to be lower or even 0. In this case

$$\Delta^2 = \frac{(S_1+S_2)^2}{2} + \frac{(S_1-S_2)^2}{2} = S_1^2(1+S_2^2/S_1^2) \tag{14.10}$$

It is clear that the combination of high and low signal-to-noise ratios would be *a better strategy* as long as $S_2^2/S_1^2 > (1-\rho)/(1+\rho)$. For example, if $\rho > 1/3$, then choosing carriers from different frequency bands with even half the signal-to-noise ratio results in better hypothesis testing. □

14.6.2 Multi-Channel Template Attacks

Just as the template attack is the optimal attack strategy for the single-channel case, a multi-channel template attack is the optimal strategy for the multi-channel case. Expanding the template approach to multiple channels is straightforward. For multiple channels, the template attack is identical except that the signals from the multiple channels are concatenated together to yield a larger signal, i.e., for each invocation, a combined signal is created by concatenating the signals from the individual observed channels. Notice that the process of identifying the time instances and sample points could end up selecting somewhat different time slices for each channel, depending purely on the nature of leakage in each channel. The maximum likelihood testing will pick up information from all channels (possibly at different times) for classification.

To show that multiple channels help the classification process, we invoke an operation on the smart card S with two different input bytes and look at just three cycles during which the input was first processed. We collected EM and power samples simultaneously and evaluated how well the template attack could classify a single EM/power trace into the two hypotheses H0 and H1 for the input byte. We did this classification first using exactly one of the power/EM channels and then performed the classification using both channels simultaneously. Figure 14.7 shows the mean EM and power signals for these hypotheses during

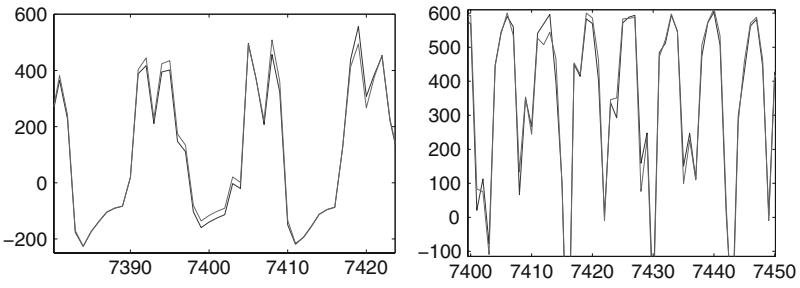

Fig. 14.7 Mean power and EM signals during three cycles for two hypothesis.

Table 14.7 Signal classification error using power, EM, and combination of power and EM.

Correct hypothesis	Error (Pwr)	Error (EM)	Error (EM+Pwr)
H0	9.5%	15.1%	2.8%
H1	20.1%	15.2%	6.6%

these three cycles side by side.[3] Table 14.7 shows the error rate of our classification effort for inputs belonging to each hypothesis. One can clearly notice that using both channels simultaneously results in better classification compared to any single channel.

14.6.3 Multi-Channel DPA

Multi-channel DPA attack is a generalization of the single-channel DPA attack. In this case, the adversary collects N signals, $\mathbf{O}_i, i = 1, \ldots, N$. In turn, each of the signals $\mathbf{O}_i$ is a collection of L signals collected from L side-channels. Thus, $\mathbf{O}_i = [\mathbf{O}_i^1, \ldots, \mathbf{O}_i^L]^T$, where $\mathbf{O}_i^l$ represents the ith signal from the lth channel. Note that all DPA-style attacks treat each time instant independently and leakages from multiple channels can only be pooled together if they occur at the same time. Thus, in order for multi-channel DPA attacks to be effective, the selected channels must have very similar leakage characteristics.

The formulae for computing the metric for multi-channel DPA attack are generalizations of those for the single channel as described in Section 14.5. The main difference is that the expected value of sample mean difference at time j under hypothesis H is a vector of length L, with the lth entry being the sample mean difference of the lth channel. Furthermore, the variance of the b-bin under hypothesis H at time j is a covariance matrix of size $L \times L$ with the i, jth entry being the correlation between signals from the ith and jth channels. Once again, as in the DPA attack, the adversary does not have the luxury of estimating these parameters. Therefore, we substitute sample estimates for these parameters along the same lines as in Equation (14.7). We skip the cumbersome formulae and directly go to the results of multi-channel DPA attacks.

Table 14.8 shows sample results of an attack on the s-box lookups in a DES implementation using the power channel together with an EM channel whose leakage is similar to the power channel. The first column shows the bit being predicted, the second shows the number of signals required for the correct key hypothesis to emerge as the winner using both channels with the multi-channel metric, the last two columns show the number of signals needed for the power and EM channels separately using the new DPA/DEMA metric. From this it is clear that the number of invocations needed for two-channel attacks can be significantly less compared to single-channel attacks.

[3] The slight offset in time is due to delay of EM signals with respect to the power signal.

Table 14.8 Multi-channel DPA-style attack using power, EM, and power and EM.

S-box hyp.	Min samples (Pwr+EM)	Min samples (Pwr)	Min samples (EM)
S1,B1	150	170	640
S1,B2	60	(>1000)	340
S1,B3	110	350	160
S2,B2	30	50	230
S2,B3	120	210	340
S4,B0	60	200	340
S6,B1	180	180	190
S7,B3	30	40	520
S8,B3	60	90	140

14.7 Toward Information Leakage Assessment

In this section, we address the following question: Can the information obtained by combining leakages from several (or even all possible) signals from available sensors be quantified regardless of the signal processing capabilities and computing power of an adversary?

We will use maximum likelihood testing to craft a methodology to assess information leakage from elementary operations in a device. This methodology takes into account the power signals and all EM signals extractable from all the given sensors across the entire EM spectrum. Results of such an assessment will enable one to bound the success probability of the optimal adversary for any given hypothesis.

Assume that for a single invocation, the adversary captures the power signal and emanations across the entire electromagnetic spectrum from all sensors in an observation vector $\mathbf{O}$. Let Ω denote the space of all possible observation vectors $\mathbf{O}$. Since the likelihood ratio, $\Lambda(\mathbf{O})$, is a function of the random vector $\mathbf{O}$, the best achievable success probability, P_s, is given by

$$P_s = \sum_{\mathbf{O}\in\Omega} I_{\{\Lambda(\mathbf{O})>1\}} p_{\mathbf{N1}}(\mathbf{O}-\mathbf{S}_1) + I_{\{\Lambda(\mathbf{O})<1\}} p_{\mathbf{N0}}(\mathbf{O}-\mathbf{S}_0) \tag{14.11}$$

where I_A denotes the indicator function of the set A and $p_{\mathbf{N1}}(\mathbf{O}-\mathbf{S}_1)$ and $p_{\mathbf{N0}}(\mathbf{O}-\mathbf{S}_0)$ are noise distributions under the hypothesis 1 and 0, respectively.

When the adversary has access to multiple invocations, an easier way of estimating the probability of success/error involves a technique based on moment generating functions. We begin by defining the logarithm of the moment generating function of the likelihood ratio:

$$\mu(s) = \ln\Big(\sum_{\mathbf{O}\in\Omega} p^s_{\mathbf{N1}}(\mathbf{O}-\mathbf{S}_1) p^{1-s}_{\mathbf{N0}}(\mathbf{O}-\mathbf{S}_0)\Big) \tag{14.12}$$

The following is a well-known result from information theory:

Fact 2 *Assume we have several statistically independent observation vectors*[4]

$$\mathbf{O}_1, \mathbf{O}_2, \ldots, \mathbf{O}_L$$

For this case, the best possible exponent in the probability of error is given by the Chernoff information*:*

$$C \stackrel{\text{def}}{=} -\min_{0 \le s \le 1} \mu(s) \stackrel{\text{def}}{=} -\mu(s_m) \tag{14.13}$$

Note that $\mu(\cdot)$ is a smooth, infinitely differentiable, convex function and therefore it is possible to approximate s_m by interpolating in the domain of interest and finding the minima. Furthermore, under certain mild conditions on the parameters, the error probability can be approximated by

$$P_\varepsilon \approx \frac{1}{\sqrt{8\pi L \mu''(s_m)} s_m (1 - s_m)} \exp(L\mu(s_m)) \tag{14.14}$$

Note that in order to evaluate Equation (14.11) or (14.14), we need to estimate $p_{\mathbf{N0}}(\cdot)$ and $p_{\mathbf{N1}}(\cdot)$. In general, this can be a difficult task. However, by exploiting certain characteristics of the CMOS devices, estimation of $p_{\mathbf{N0}}(\cdot)$ and $p_{\mathbf{N1}}(\cdot)$ can be made more tractable.

14.7.1 Practical Considerations

We will now outline some of the practical issues associated with estimating $p_{\mathbf{N0}}(\cdot)$ and $p_{\mathbf{N1}}(\cdot)$ for any hypothesis. The key here is to estimate the noise distribution for each cycle of each elementary operation and for each relevant state R that the operation can be invoked with. This results in the signal characterization, $\mathbf{S}_R$, and the noise distribution, $p_{\mathbf{NR}}(\cdot)$, which is sufficient for evaluating $p_{\mathbf{N0}}(\cdot)$ and $p_{\mathbf{N1}}(\cdot)$.

There are two crucial assumptions that facilitate estimating $p_{\mathbf{NR}}(\cdot)$: first, on chip-cards examined by us the typical clock cycle is 270 ns. For such devices, most of the compromising emanations are well below 1 GHz which can be captured by sampling the signals at a Nyquist rate of 2 GHz. This sampling rate results in a vector of 540 points per cycle per sensor. Alternatively, one can also capture all compromising emanations by sampling judiciously chosen and slightly overlapping bands of the EM spectrum. The choice of selected bands is dictated by considerations such as signal strength and limitations of the available equipment. Note that the slight overlapping of EM bands would result in a corresponding increase in the number of samples per clock cycle; however, it remains in the range of 600–800 samples per sensor.

[4] For simplicity, this chapter deals with *independent* elementary operation invocations. Techniques also exist for adaptive invocations.

The second assumption, borne out in practice (see [6]), is that for a fixed relevant state, the noise distribution $p_{\mathbf{NR}}(\cdot)$ can be approximated by a Gaussian distribution. This fact greatly simplifies the estimation of $p_{\mathbf{NR}}(\cdot)$ as only about one thousand samples are needed to roughly characterize $p_{\mathbf{NR}}(\cdot)$. Moreover, the noise density can be stored compactly in terms of the parameters of the Gaussian distribution.

These two assumptions imply that in order to estimate $p_{\mathbf{NR}}(\cdot)$ for a fixed relevant state R, we need to repeatedly invoke (say 1000 times) an operation on the device starting in the state R and collect samples of the emanations as described above. Subsequently, the signal characterization S_R can be obtained by averaging the collected samples. The noise characterization is obtained by first subtracting S_R from each of the samples and then using the Gaussian assumption to estimate the parameters of the noise distribution.

The assessment can now be used to bound the success of any hypothesis testing attack in our adversarial model. For any two given distributions B_0 and B_1 on the relevant states, the corresponding signal and noise characterizations

$$S_0, S_1, p_{\mathbf{N0}}(\cdot), \text{ and } p_{\mathbf{N1}}(\cdot)$$

are a *weighted sum* of the signal and noise assessments of the constituent relevant states S_R and $p_{\mathbf{NR}}(\cdot)$. The error probability of maximum likelihood testing for a single invocation or its exponent for L invocations can then be bounded using Equations (14.11) and (14.13), respectively.

We now give a rough estimate of the effort required to obtain the leakage assessment of an elementary operation. The biggest constraint in this process is the time required to collect samples from approximately 1000 invocations for each relevant state of the elementary operation. For an r-bit machine, the relevant states of interest are approximately 2^{2r}; thus, the leakage assessment requires time to perform approximately $1000 * 2^{2r}$ invocations. Assuming that the noise is Gaussian and that each sensor produces an observation vector of length 800, for n sensors the covariance matrix Σ_N has $(800 * n)^2$ entries. It follows that the computation burden of estimating the noise distribution would be proportional to $(800 * n)^2$. Such an approach is certainly feasible for an evaluation agency from both a physical and computational viewpoint, as long as the size of the relevant state, r, is small.

14.8 Projects

Pre-requisite: Exercises 1–4: A DPA setup for smart cards and sample smart cards. Exercises 1, 2, and 3 require the following data collection:

Capture a set S_A of a few thousand power signals from a smart card with DPA-countermeasures turned off, operating on some fixed input A. Capture another set of signals S_B (where $|S_B| = |S_A|$) from the same smart card operating with another fixed input B, which is very similar to A (e.g., A and B could differ only on one

bit). Partition the sets S_A and S_B into two equal-sized training sets S_{A1} and S_{B1} and two equal-sized testing sets S_{A2} and S_{B2}, such that S_{A2} and S_{B2} contain at least a few hundred signals each.

1. *Single Point Binary Classification:* Compute the mean signals μ_A and μ_B using the sets S_{A1} and S_{B1}. Subtract μ_A from μ_B and plot the difference of means signal to locate the first point in time P where there is a significant difference (or peak). At this point P, signals in S_{A1} have a mean μ_{AP} and signals in S_{B1} have a mean μ_{BP}. Use the set S_{A1} to also compute the variance σ^2_{AP} of these signals at point P and likewise compute σ^2_{BP} from S_{B1}. Make the assumption that the signals from S_{A1} and S_{B1} are normally distributed at point P (Gaussian assumption, with $n = 1$). Next use the maximum likelihood principle and Gaussian assumption about point P to classify signals from testing sets S_{A2} and S_{B2} by just looking at point P. Compute the fraction of correct/incorrect classification from sets S_{A2} and S_{B2}.
2. *Multi-Point Binary Classification, Univariate Statistics:* Build upon experiment 1 above by considering two other peaks that are located near the first peak P. In this exercise, compute the means and variances at these three points for inputs A and B by using the training sets S_{A1} and S_{B1}. Then use the maximum likelihood principle with univariate Gaussian statistics (i.e., assuming that the noise co-variance matrix across these three points is diagonal) to classify the test signals from S_{A2} and S_{B2} using these means and variances only. Compute the fraction of correct/incorrect classification from sets S_{A2} and S_{B2}.
3. *Multi-Point Binary Classification, Multivariate Statistics:* Repeat experiment 2 above with multivariate statistics, i.e., by computing the covariance between the noise at the three different points in time and using the maximum likelihood principle and Gaussian assumption to classify the test signals. Compute the fraction of correct/incorrect classification from sets S_{A2} and S_{B2}. Compare the results of experiments 1, 2, and 3 to see how adding additional information (points) and better analysis (multivariate statistics) improves the classification accuracy.
4. *Comparison of Metrics for DPA:* Run a DPA experiment on your smartcard with the usual DPA metric (Equation 14.5). Run the same experiment with the metric provided in Equation (14.8). Compare the results in terms of number of signals needed to obtain the correct result of the DPA analysis.
5. *Signal Selection:* Suppose you can collect two signals $[O_1, O_2]$ at a point in time (see Example 14.1) and use these two signals to determine between two hypotheses: H_0, $O_k = N_k$, for $k = 1, 2$, and under the hypothesis H_1, $O_k = S_k + N_k$. Assume that $\mathbf{N_i} = (N_1, N_2)^T$ has zero mean multivariate Gaussian distribution with

$$\Sigma_N = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

Signal O_1 has already been selected and $O_1 = S_1 + N_1$. You have three choices for the second signal O_2:

1. Choice 1: $S2 = 0.9 * S1, \rho = 0.5$.
2. Choice 2: $S2 = 0.8 * S1, \rho = 0.2$.
3. Choice 3: $S2 = 0.65 * S1, \rho = 0$.

Which of these signals should you choose for O_2 and why?

References

1. O. Acıicmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In M. Abe editor, *Topics in Cryptology CT-RSA 2007, The Cryptographers Track at the RSA Conference 2007*, pp. 225–242, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007.
2. D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In B. Kaliski, Ç. K. Koç, and C. Paar editors, *Proceedings of CHES 2002*. Lecture Notes in Computer Science, vol. 2523, pp. 29–45, Springer, 2002.
3. D. Agrawal, J. R. Rao, P. Rohatgi, and K. Schramm. Templates as Master Keys. In J. R. Rao and B. Sunar editors, *Proceedings of CHES 2005*, Lecture Notes in Computer Science, vol. 3659, pp. 15–29, Springer, 2005.
4. D. Asinov and R. Agrawal. Keyboard acoustic emanations. In *Proceeding of the IEEE Symposium on Security and Privacy 2004*, pp. 3–11, 2004.
5. D. J. Bernstein. Cache-timing attacks on AES. Technical Report, p. 37, April 2005, available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
6. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. Kaliski, Ç. K. Koç, and C. Paar editors, *Proceedings of CHES 2002*, Lecture Notes in Computer Science, vol. 2523, pp. 13–28 Springer, 2002.
7. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In Ç. K. Koç, D. Naccache, and C. Paar editors, *Proceedings of CHES 2001*, Lecture Notes in Computer Science, vol. 2162, pp. 251–261, Springer, 2001.
8. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz editor, *Advances in Cryptology – CRYPTO '96*, Lecture Notes in Computer Science, vol. 1109, pp. 104–113, Springer-Verlag, 1996.
9. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener editor, *Proceedings of Advances in Cryptology CRYPTO '99*, Lecture Notes in Computer Science, vol. 1666, pp. 388–397, Springer-Verlag, 1999.
10. M. Kuhn. Optical Time-domain eavesdropping risks of CRT displays. In *Proceedings of the Symposium on Security and Privacy*, pp. 3–18, 2002.
11. J. Loughry and D. Umphress. Information leakage from optical emanations. In *ACM Transactions on Information and System Security*, vol. 5, pp. 262–289, 2002.

12. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval editor, *Topics in Cryptology CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*, pp. 1–20, Lecture Notes in Computer Science, vol. 3860, Springer-Verlag, 2006.
13. C. Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, 2005, available at http://www.daemonology.net/hyperthreading-considered-harmful/
14. J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and countermeasures for smart cards. In *Proceedings of e-Smart 2001*, Lectures Notes in Computer Science (LNCS), vol. 2140, pp. 200–210, Springer, 2001.
15. H. L. Van Trees. *Detection, Estimation, and Modulation Theory*, Part I. John Wiley & Sons, New York, 1968.

Chapter 15
Electromagnetic Attacks and Countermeasures

Pankaj Rohatgi

15.1 Introduction and History

EM is a side-channel with a long history of rumors and leaks associated with its use for espionage. It is well known that defense organizations across the world are paranoid about limiting EM emanations from their equipment and facilities and conduct research on EM attacks and defenses in total secrecy. In the United States, such work is classified under the codename "TEMPEST" which is believed to be an acronym for "transient electromagnetic pulse emanation standard". In January 2001, in response to a Freedom of Information Act (FOIA) request, some documents related to TEMPEST such as *NACSIM 5000 tempest fundamentals*, *NACSEM 5112 NONSTOP evaluation techniques* and *NSTISSI no. 7000 TEMPEST countermeasures for facilities* were released in redacted form and can be downloaded from the website http://www.cryptome.org.

In the public domain, the significance of the EM side-channel was first demonstrated by van Eck in 1985 [11]. He showed that EM emanations from computer monitors could be captured from a distance and used to reconstruct the information being displayed. Figures 15.1 and 15.2 show a modern day recreation of this attack, where the contents of the computer monitor displaying a Word document in Figure 15.1 have been reconstructed in Figure 15.2 using only the EM emanations from that monitor. As a defense against this attack, Kuhn and Anderson in 1998 [8] developed special fonts which have substantially reduced EM leakage characteristics which make them difficult to reconstruct.

The first openly published works on EM analysis of ICs and CPUs performing cryptographic operations by Quisquater and Samyde [9] and by Gandolfi, Mourtel and Olivier [5] in 2001 were quite limited. These attacks were performed on chip cards and required tiny antennas to be placed in very close proximity to the IC being attacked. In fact, the best attacks were semi-invasive, requiring the decapsulation

IBM T. J. Watson Research Center
e-mail: rohatgi@us.ibm.com

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_15,

© Springer Science+Business Media, LLC 2009

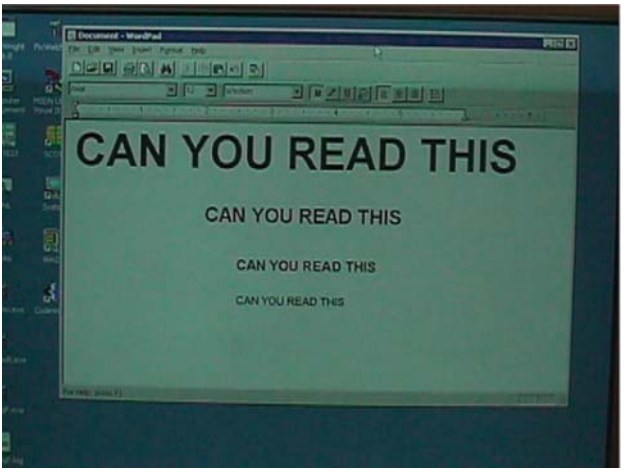

Fig. 15.1 Computer display.

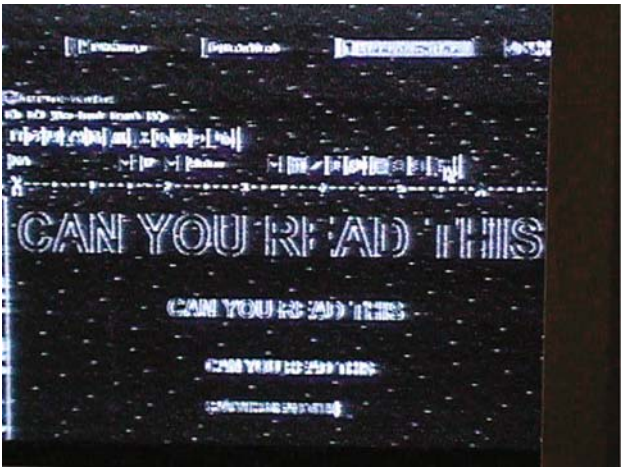

Fig. 15.2 Computer display reconstructed from EM.

of the chip packaging and careful positioning of micro-antennas on the passivation layer of the chip substrate to isolate the signals of interest. The EM signals were used to demonstrate attacks such as simple and differential EM analysis (SEMA/DEMA).

Subsequently the work of Agrawal, Archambeault, Rao and Rohatgi in 2002 [1], which was much closer to the declassified TEMPEST literature, removed these limitations and showed that EM attacks on CPUs and cryptographic devices were possible at a distance and that the EM side-channel leaks information that is not easily available from the power side-channel. This work included a systematic study of EM leakages from computing equipment and peripherals, such as chip cards, CPUs, crypto accelerators, monitors, keyboards and peripherals, comparison of the EM

side-channel to other side-channels and a methodology for leakage assessment. This work has appeared in cryptology ePrint archives May 2001, CHES 2002, CHES 2003, RSA Labs CryptoBytes Spring 2003 and forms the basis for this chapter.

15.2 EM Emanations Background

A deep understanding of the different types of EM leakages and the propagation of EM signals are essential in order to conduct EM side-channel attacks and to develop techniques to defend against such attacks.

Some of the earlier published work on EM emanations focused on one particular form of EM leakage, i.e., the *direct emanations* from chip cards and good quality direct EM emanations turned out to be very hard to capture without invasive techniques and careful micro-antenna positioning. In reality, once the different forms of EM emanations are understood, there are usually several possible EM signals that can be easily captured from a device and used for EM analysis. In fact a single EM sensor may be able to multiply EM signals even from a distance. This fact is succinctly captured in the following quote from the NASCIM 5000 Tempest Fundamentals document.

> "The forms in which compromising emanations might appear at an interception point are numerous."

15.2.1 Types of EM Emanations

There are two broad classes of EM emanations:

1. Direct Emanations: These emanations result from *intentional* current flows within circuits. These generate time-varying electric and magnetic fields related by Maxwell's equations. In CMOS circuits, these current flows consist of short bursts of current with sharp rising edges that occur during the switching operation and result in EM emanations observable over a wide frequency band. Often, higher frequency emanations are more useful to the attacker since there is substantial noise and interference in the lower frequency bands. In complex circuits, it may be quite difficult to isolate direct emanations due to interference from other signals. Reducing such interference requires tiny probes positioned very close to the signal source and/or special filters to separate the desired signal from other interfering signals.

The initial published work on EM analysis by Quisquater and Samyde [9] and Gandolfi, Mourtel and Olivier [5] focused exclusively on direct emanations, in particular they focused on using tiny coils to capture the time-varying magnetic fields created by intentional currents.

2. Unintentional Emanations: Most modern devices pack a large number of circuits and components into a very small area and suffer from numerous unintentional electrical and electromagnetic couplings between components, depending on their proximity and geometry. The vast majority of these couplings are minor and are

ignored by circuit designers since they do not affect functionality. Such couplings, however, are a rich source of compromising emanations. These emanations manifest themselves as **modulations** *of carrier signals* generated, present or introduced within the device. Depending on the type of coupling, the carrier can be *amplitude modulated* or *angle modulated* by the sensitive signal, or the modulation could be more complex. If a modulated carrier can be captured, the sensitive signal can be recovered by an EM receiver tuned to the carrier frequency and performing the appropriate demodulation.

The various types of EM emanations are succinctly described in the following quotes from NACSIM 5000 Tempest Fundamentals document:

> "The strongest and most numerous electromagnetic emanations are generated by sharp-rising and current waveforms of short duration ⋯. Also, faster rise times generate additional emanations – harmonics – of progressively lower amplitudes from the same pulse source, these harmonics ⋯ represent, in effect, a great many compromising signals. These signals can be acquired not only by being correctly tuned to the fundamental frequency, but also at any of the harmonic frequencies ⋯ . At times, in fact, harmonics are more useful than the fundamental, i.e., Emanations at the fundamental frequency are often lost among other signals of the same frequency, whereas a harmonic might be more easily isolated."
>
> ⋮
>
> "Modulated spurious carriers (U). - This type of CE is generated as the modulation of a carrier by RED data. ⋯. The carrier is usually amplitude or angle-modulated by the basic red data signal. Or a signal related to the basic RED data signal, which is then radiated into space or coupled into EUT external conductors."

Exploiting direct emanations requires close physical proximity to be effective. In contrast, unintentional emanations are usually much easier to capture and exploit since some modulated carriers are much stronger and propagate much further than direct emanations. This enables attacks to be carried out at a distance without resorting to any invasive techniques. Rich sources of such carriers include the periodic, harmonic-rich clock signal(s) and signals used for internal and external communication. For example, an ideal, symmetric, "square-wave" clock signal depicted in Figure 15.3, when viewed in the frequency domain in Figure 15.4, consists of a dominant component at the fundamental frequency together with components at all the *odd* harmonics with linearly decreasing amplitude. In practice, the actual clock signal is far from ideal and usually contains a limited number of significant odd harmonics and some even harmonics as well.

15.2.2 EM Propagation

EM emanations can propagate both via radiation and via conduction. Often, EM emanations arrive at an intercept point by a complex combination of radiation and conduction. This phenomenon is well described in the following quotes from NACSIM 5000 Tempest fundamentals:

> Propagation of EM Emanations
>
> "Modulated spurious carriers (U). - This type of CE is generated as the modulation of a carrier by RED data. ⋯ The carrier is usually amplitude or angle-modulated by the basic

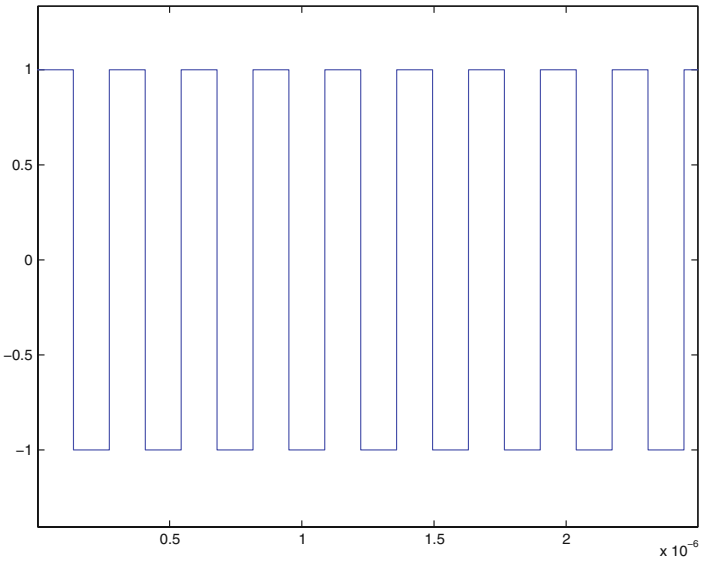

Fig. 15.3 Ideal clock signal.

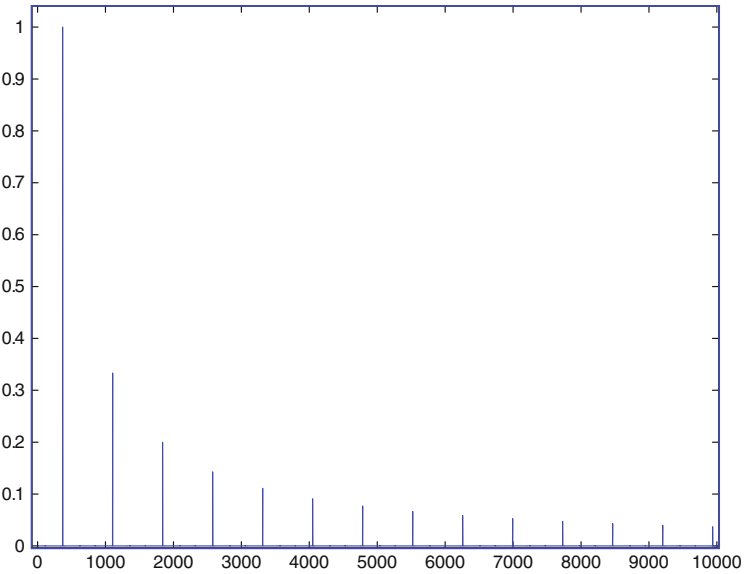

Fig. 15.4 FFT of an ideal clock: $s(t) = \frac{4}{\pi}\Sigma_{n=1,3,5,\ldots,} \frac{1}{n}\sin(n\omega t)$.

red data signal. Or a signal related to the basic RED data signal, which is then radiated into space or coupled into EUT external conductors."

...

"There are four basic means by which compromising emanations may be propagated. They are: electromagnetic radiation; conduction; modulation of an intended signal; and acoustics.

> A brief explanation of each follows. a. (C) Electromagnetic Radiation (U). - Whenever a RED signal is generated or processed in an equipment, an electric, magnetic or electromagnetic field is generated. If this electromagnetic field is permitted to exist outside of an equipment, a twofold problem is created; first the electromagnetic field may be detected outside the Controlled Space (CS); second the electromagnetic field may couple onto BLACK lines connected to or located near the equipments, which exit the CS of the installation. b. (C) Line Conduction. - Line Conduction is defined as the emanations produced on any external or interface line of an equipment, which, in any way, alters the signal on the external or interface lines. The external lines include signal lines, control and indicator lines, and a.c. and d.c. powerlines. c. (C) Fortuitous Conduction. - Emanations in the form of signals propagated along any unintended conductor such as pipes, beams, wires, cables, conduits, ducts, etc. d. (C) [Six lines redacted.]"

From an attacker's perspective, conducted emanations are more useful than radiated emanations. Radiated emanations attenuate rapidly with distance and need to be captured close to the device since they obey the inverse square law. Conducted emanations attenuate linearly with distance and thus can be intercepted at greater distances.

The following example illustrates conducted EM emanations. Currents on the power line of smart cards have been well studied in the context of power analysis. For example, Figure 15.5 shows the amplitude of the current flowing on the power line of a smart card while it is performing three rounds of DES. This fact is clearly visible in the power signal which shows a basic signal shape for a DES round that is repeated three times during this time window. Now the power line is also a conductor and therefore is likely to carry conductive EM emanations as well. The faint, AM-modulated EM signals at low carrier frequencies are overwhelmed by larger power-

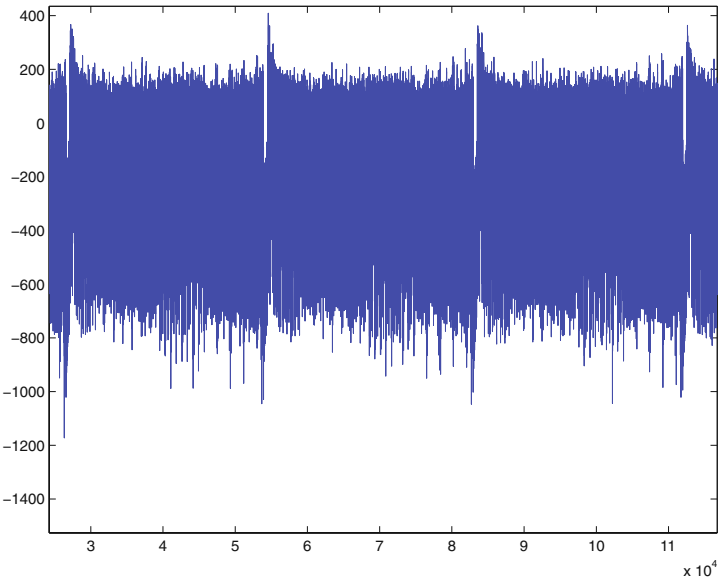

Fig. 15.5 Raw power signal during three rounds of DES.

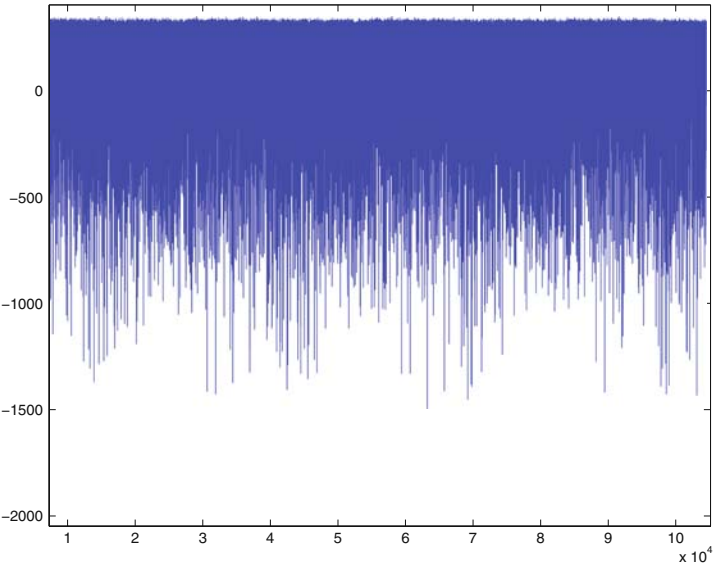

Fig. 15.6 Conducted EM signal on the power line during three rounds of DES.

consuming currents, but the faint, AM-modulated EM signals at higher carrier frequencies can be easily separated and demodulated to yield compromising information. Figure 15.6 shows the demodulated EM signal obtained from the power line, which also displays a (different) basic shape for a DES round repeated three times.

15.3 EM Capturing Equipment

Like power analysis, an EM attack system requires sample collection equipment such as a digital oscilloscope or a sampling board as well as software for controlling device operations, triggering and controlling data collection and for signal processing and analysis.

Radiated EM signals in the near field can be captured using near-field probes. Signals in the far field can be captured by antennas appropriate for the band being considered. Antennas and probes are not expensive and can even be constructed at low cost. Conducted emanations on the power or ground lines are best captured using LISNs (line impedance stabilization networks) and signals from fortuitous conductors can be processed directly.

The *critical* piece of equipment for performing EM attacks is a tunable receiver/demodulator which can be tuned to various modulated carriers and can perform demodulation to extract the sensitive signal. High-end receivers such as the Dynamic Sciences R-1550 (see [4]) are ideal for this purpose since they cover a wide band and offer a large selection of bandwidths and demodulation options. However, wideband/wide-bandwidth receivers tend to be quite expensive even when

Fig. 15.7 A second-hand wideband, wide-bandwidth receiver.

Fig. 15.8 ICOM 7000 receiver.

purchased second-hand (see Figure 15.7). Another option is to use certain wideband radio receivers that provide a large bandwidth intermediate frequency (IF) output in addition to the audio output. One such receiver is the ICOM 7000 (see Figure 15.8) which can be purchased second-hand for less than $1000. The IF output can be sampled and demodulated by software to extract the signal. However, such receivers introduce significant noise into the captured signals and are not suitable for capturing very faint signals that are close to the thermal noise floor. In addition, these receivers only provide a few MHz of bandwidth which is not enough to capture the internals of devices operating at high frequencies. Those on low budgets can construct their own low-noise receiver for under $1000 by using commonly available low-noise electronic components (see Figure 15.9), common lab equipment and demodulation software, but this approach can become inconvenient due to the need for frequent calibration. However, once the best signal to attack is identified, a custom, non-tunable receiver/demodulator for the attack can be built quite cheaply.

Common laboratory equipment such as spectrum analyzers are also very useful for quickly assessing the available EM signals to identify potentially useful carriers.

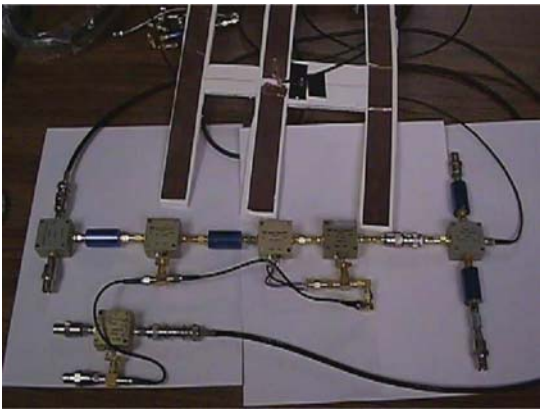

Fig. 15.9 Low-cost, low-noise receiver built from components.

15.4 EM Leakage Examples

In this section we will describe several experiments which illustrate the types of EM signals and EM side-channels available from several different devices and describe possible avenues for attack.

15.4.1 Examples: Amplitude Modulation

In our first set of experiments, we will explore EM side-channels available via amplitude demodulation of a carrier signal. Our first example is a 6805-based smart card operating on a 3.68 MHz external clock and performing the following set of three instructions continuously in a 13-cycle loop:

1. Access RAM containing a value B (5 cycles)
2. Check for external condition (5 cycles)
3. Jump back to start of loop (3 cycles)

Figure 15.10 shows the raw signal obtained by a near-field EM sensor placed behind the smart card during a time interval in which the card executed around 26 cycles or 2 iterations of the loop. The figure shows a very regular signal structure repeated 26 times. On closer examination, this regular structure turns out to be the differential of the clock signal. This is not surprising since the clock is the most dominant signal and *direct emanation* within the card. From the raw signal, it is not possible to discern the fact that the smart card is operating in a loop or to know the nature of the operations being performed. This figure also highlights the problem of working with *direct emanations*. In this case, the clock signal is so dominant that information about other currents within the smart card have been washed out. Extracting these smaller signals will require careful micro-antenna positioning in close proximity to these signal sources.

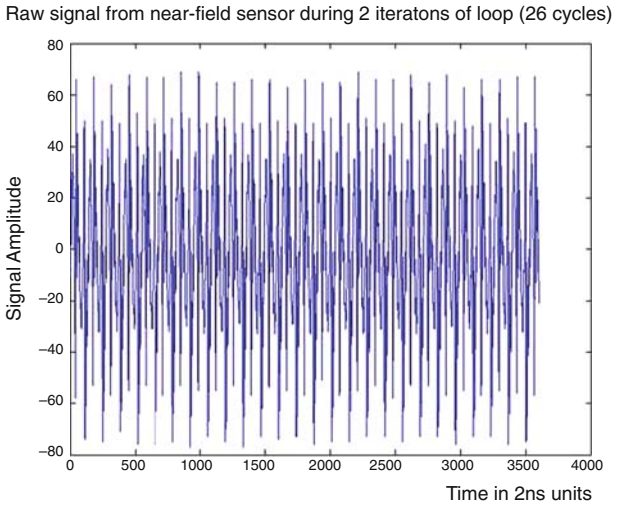

Fig. 15.10 Raw EM signal from 6805 smart card during 26 clock cycles.

This situation becomes clearer once the FFT of the raw signal is examined as shown in Figure 15.11. Here the dominant signal is the clock signal, which consists of strong components at the fundamental frequency and at odd harmonics as well as some components at even harmonics. Information about the internal operations of the smart card, such as the fact that it is operating in a loop with a frequency that is 1/13th the clock frequency, is not readily apparent in the FFT; these signals have very low amplitude and appear as noise in between the clock harmonics.

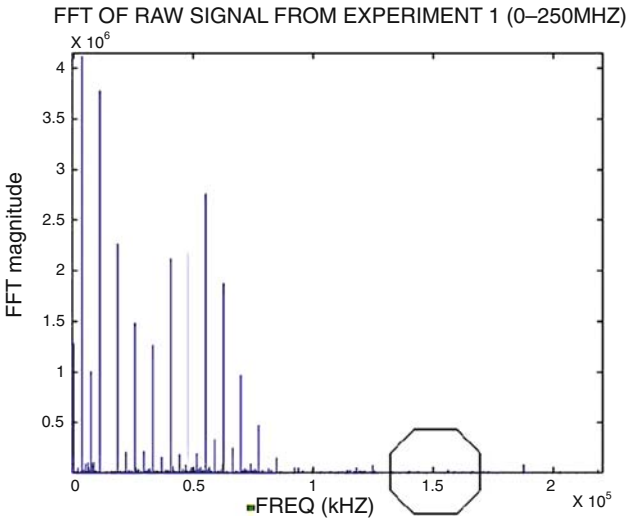

Fig. 15.11 FFT of raw EM signal from 6805 smart card.

However, at higher frequencies, say beyond 100 MHz, the amplitude of the clock harmonics have been significantly reduced and these smaller signals can be extracted via AM demodulation by tuning a receiver at one of these clock harmonics. Figure 15.12 shows the result of AM demodulating the raw signal at the 41st clock harmonic with a center frequency of around 150 MHz. The demodulated signal, which again covers around 26 cycles, shows the structure of the computation quite clearly. It is easy to see that these 26 cycles consist of a basic signal repeated twice, i.e., a loop of 13 cycles, and the internals of this basic signal show three different substructures of 5 cycles, 5 cycles and 3 cycles which represents the three instructions in the loop.

Just like the power side-channel, once the compromising EM signals are extracted, they provide details about the computation. For example using the same AM demodulating technique, if one looks at the same smart card performing DES, at a large time scale (see Figure 15.13) one can discern the 16 rounds of DES; at an intermediate time scale (see Figure 15.14) one can discern the internals of the computation during two rounds of DES; and at a very fine time scale (see Figure 15.15) one can get information at the clock cycle level.

Our second example is a Palm Pilot which has been loaded with software developed by Feng Zhu of Northeastern University to perform elliptic curve cryptography. In particular it has been programmed to perform the point multiplication operation kP where P is a point on a Koblitz curve over GF$[2^{163}]$. The multiplication operation is performed using Solinas's technique which replaces the traditional point doubling operation by the highly efficient Frobenius map (τ) computation as follows:

- First the secret k is decomposed into its τ-adic NAF (non-adjacent form), i.e., $k = \Sigma s_i \tau^i$ where $s_i \in 0, 1, -1$ and no two adjacent s_i's can be nonzero.

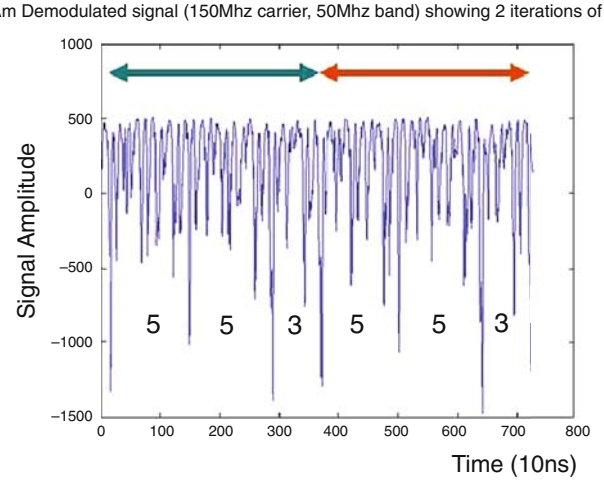

Fig. 15.12 Demodulated EM signal from 6805 smart card during 26 clock cycles.

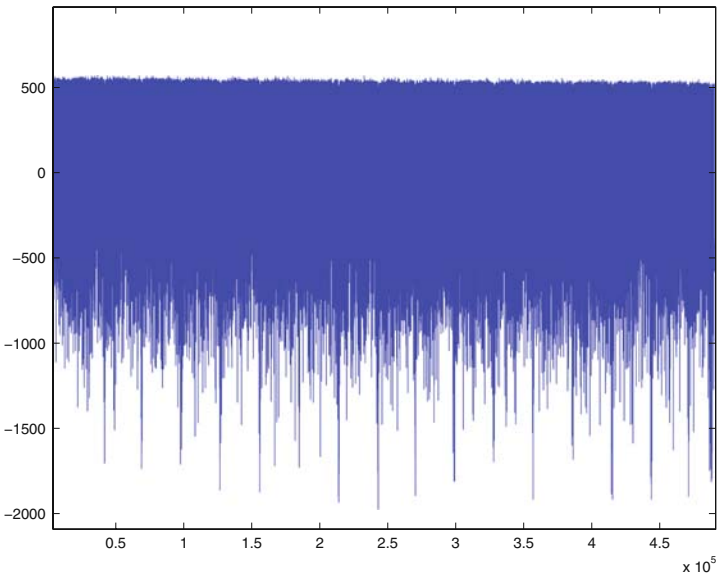

Fig. 15.13 Demodulated EM signal (100 MHz bandwidth) from smart card performing 16 rounds of DES.

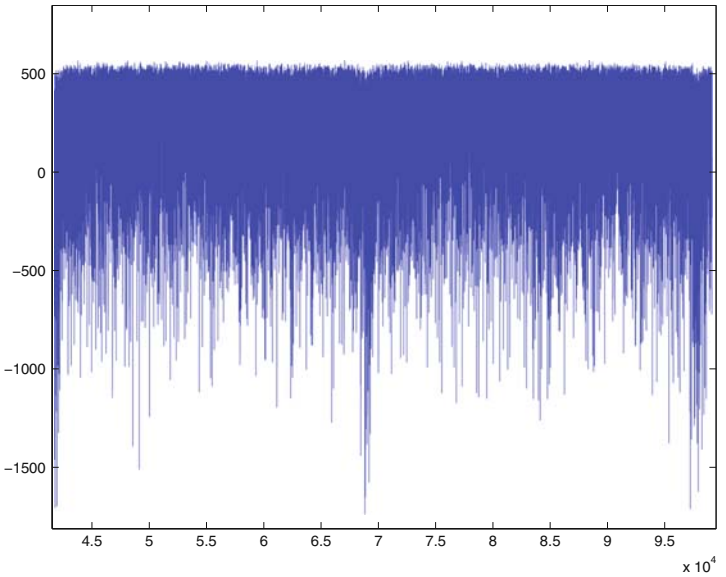

Fig. 15.14 Demodulated EM signal showing two rounds of DES (100 MHz bandwidth).

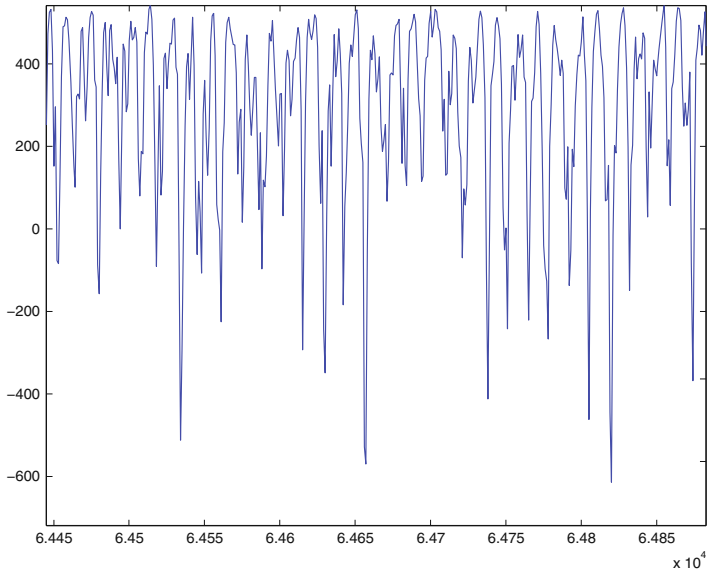

Fig. 15.15 Demodulated EM signal: clock cycle level details within a DES round.

- The traditional double/add algorithm is replaced by an algorithm that performs a sequence of τ-transforms followed by add/subtract based on the value of s_i.

The advantage of this technique is that the computational cost of the kP operation is approximately $|k|/3 \approx 54$ point additions/subtractions, since the τ-transform operation is very efficient.

The EM emanations from the Palm Pilot can be picked up even a few centimeters away from the device. A fairly good signal showing internal operations is available via AM demodulation at 241 MHz. The signal shown in Figure 15.16 immediately provides the sequence of τ-transforms (where s_i is 0) and the add/subtract operations (where s_i is $+1$ or -1). Recovering the key k further requires distinguishing between the add and subtract operations, but as Figure 15.17 shows, under intermediate level of resolution these operations are distinct. Thus we have a simple electromagnetic attack (SEMA) against this implementation.

Our final example for AM demodulation is a PCI bus-based RSA accelerator S inside a Intel/Linux server. Multiple AM-modulated carriers are available from that device, mostly at odd harmonics of the PCI clock of 33 MHz. Several carriers from this device propagate upto 50 feet and through walls enabling precise RSA timing to be measurable from around 50 feet. This precise timing could be used to perform better timing attacks than via remote interaction with the server. In addition to high-energy carriers at multiples of the PCI clock frequency, there were also several intermediate strength *intermodulated* carriers at other frequencies. These intermodulated carriers arise due to nonlinear interactions among the various carriers present within the accelerator's operating environment. These carriers provided more details

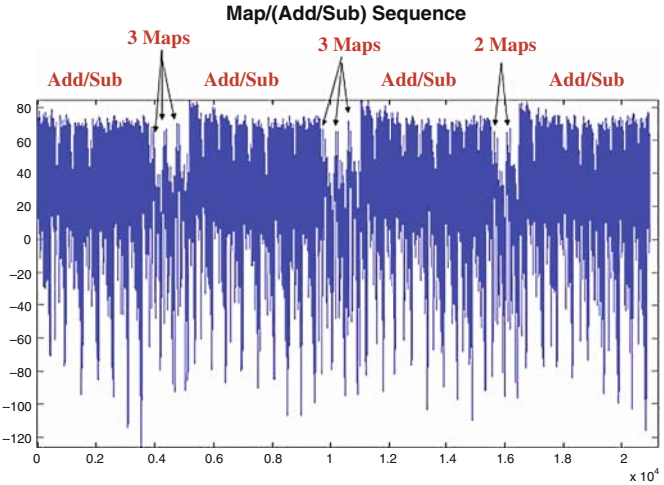

Fig. 15.16 EM signal from Palm Pilot showing elliptic curve operation sequence.

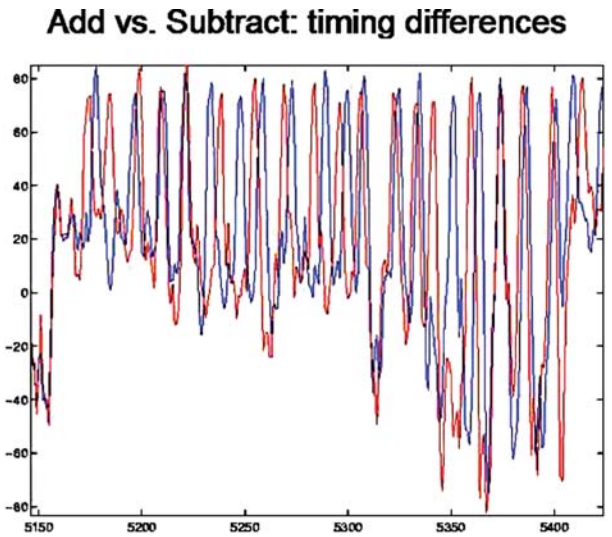

Fig. 15.17 EM signal from Palm Pilot: add vs. subtract.

of the internals of the RSA operation in S. For example, AM demodulating an intermodulated carrier at 461.4 MHz provided detailed information even from 3 to 4 feet away.

Figure 15.18 shows the signal obtained by AM demodulating the 461.46 MHz intermodulated carrier with a band of 150 KHz for a period of 2.5 ms during which S computes two successive and identical 2048-bit modular exponentiations with a 12-bit exponent. For clarity, the figure shows an average taken over 10 signal samples. One can clearly see a basic signal shape repeated twice, with each repetition

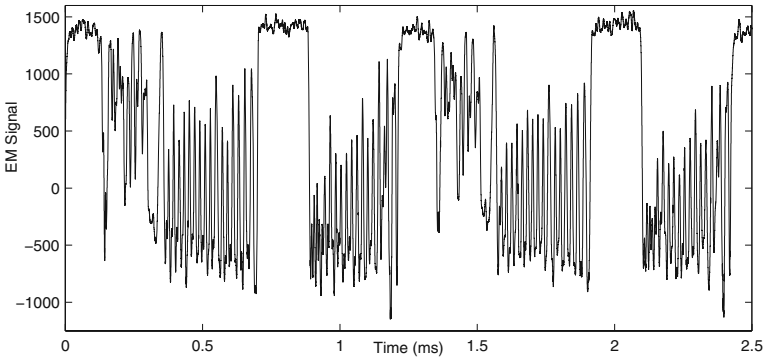

Fig. 15.18 EM signal from SSL accelerator S.

corresponding to a modular exponentiation. The first repetition spans the time interval from 0 to 1.2 ms and the second from 1.2 to 2.4 ms. The signal also shows the internal structure of the exponentiation operation. From time 0 to 0.9 ms, S receives the exponentiation request and performs some precomputation to initialize itself to exponentiate using the Montgomery method. The actual 12-bit exponentiation takes place approximately from time 0.9 to 1.2 ms. A closer inspection of this region reveals substantial information leakage which is beneficial to an adversary. Figure 15.19 plots an expanded view of this region for two different exponentiation requests which have the same modulus and exponent but different data. The two signals are plotted in different line styles (solid and broken). From the start, one can see that the two signals go in and out of alignment due to data-dependent timing of the Montgomery multiplications employed by this implementation. This data dependence of the Montgomery multiplication operation provides the basis for most of the attacks against S (see [2], [10] and [12]).

At intermediate distances of 10–15 feet, the level of noise increases significantly, but simple statistical attacks on S are still feasible and require a few thousand

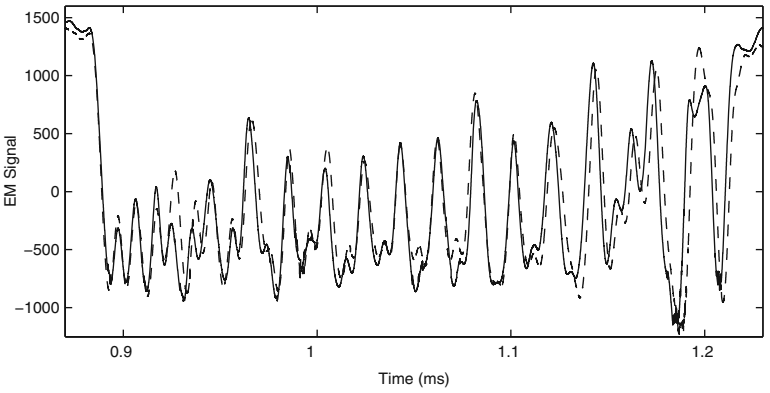

Fig. 15.19 Two EM signals, different data, same modulus, same exponent.

samples. However, attacks that are limited to one or a few samples become much harder and quickly start approaching the limits of even the advanced signal analysis techniques such as template attacks that will be described in another chapter.

15.4.2 Examples: Angle Modulation

Next we look at EM emanations that manifest as angle modulations of a carrier signal. Our first example is the same 6805-based smart card as before running the same 13-cycle loop, i.e.,

1. Access RAM containing a value B (5 cycles)
2. Check for external condition (5 cycles)
3. Jump back to start of loop (3 cycles)

but now the smart card is run on its internally generated, variable clock. In this case, as a DPA countermeasure, the clock is designed to run freely with its frequency changing with time. The smart card was tested with different values of the byte B and the following behavior was observed (see Figure 15.20): When the byte B had an LSB of 0, the loop ran faster, when it was 1 the loop ran slower. This means that the internally generated clock signal is being angle modulated by the least significant bit on the bus! The clock signal being the strongest EM signal can be captured from a distance and by angle demodulating this signal one gets information about the LSB on the bus.

The second example is another PCI-based RSA/Crypto Accelerator R inside an Intel/Linux server. After AM demodulating a 99 MHz carrier (clock harmonic) some

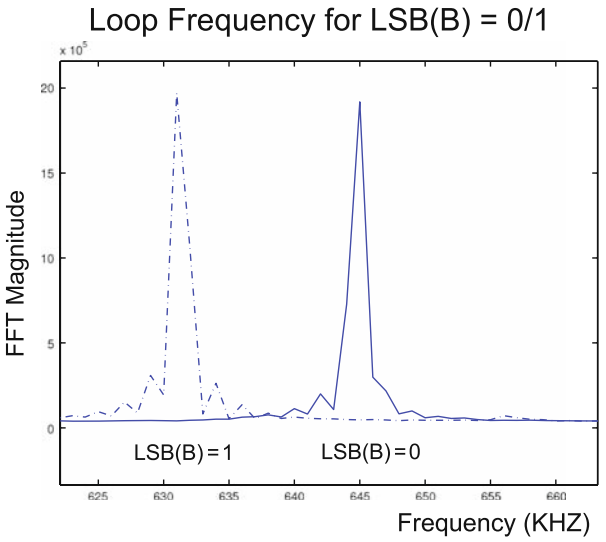

Fig. 15.20 Loop frequency related to LSB(B)!

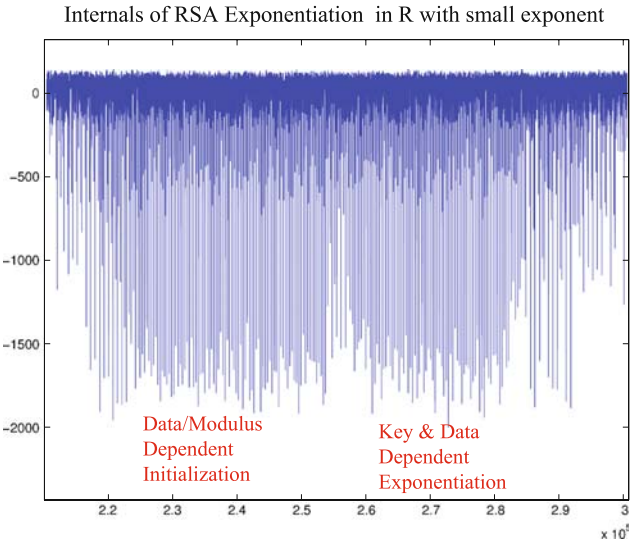

Fig. 15.21 Macro view of internal operations within Crypto Accelerator R.

information about the internal operations of R is available as shown in Figure 15.21, where the RSA operation is seen to consist of two stages: an initialization stage followed by an exponentiation stage. However, at finer time scales, the information about the internal operations of R is obscured by another, asynchronous signal G as shown in Figure 15.22. Due to this interference it appears that one may not be able to reconstruct the internals of the RSA operation to attack this device.

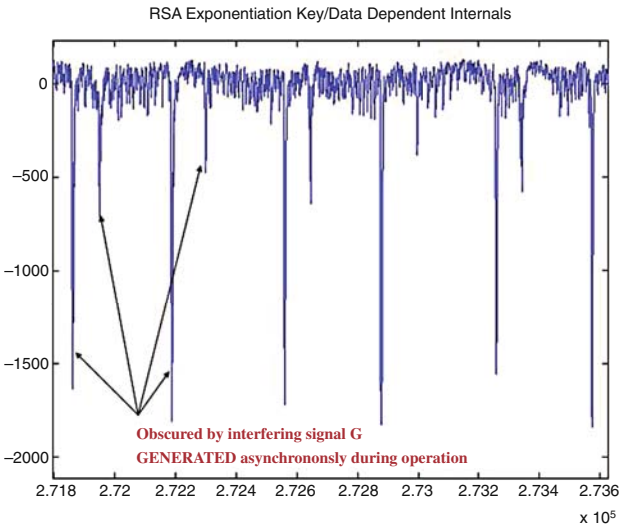

Fig. 15.22 Signal G obscures details of internals of R.

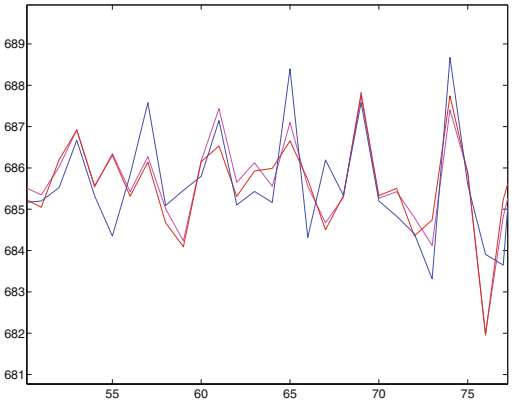

Fig. 15.23 Timing characteristics of G for three keys (two same).

But, as mentioned earlier, due to coupling effects, the timing of asynchronously generated signals is usually affected by the operations being performed within a device. This turns out to be the case for the signal G as well. Timing statistics of G (using 1000 samples) gives information about internals as can be seen in Figure 15.23, which shows the timing characteristics of G in three independent runs with three exponents, two of which are the same. This figure shows the average inter-peak time between the different peaks in G. As seen from this figure, when the keys are the same, the timing characteristics are very similar and quite different from the timing characteristics for a dissimilar key. An attacker who can get around 1000 EM samples from one device R1 can use the timing statistics of G to determine the key used by R1 if he can get access to an identical test device R2. The attacker would reconstruct the key bit-by-bit by comparing the timing statistics of the signal G for different test keys in R2 with the timing characteristics of the signal G obtained from R1. Moreover, since the signal G is strong enough to be captured even at a distance of 10–15 feet, the attack may be quite practical.

15.5 Multiplicity of EM Channels and Comparison with Power Channel

Based on the experiments described above, it is clear that there are multiple EM side-channels based on amplitude or angle demodulating different carriers which may be generated within the device, present in the environment or deliberately introduced within the device. We have also seen that often higher frequency, low-energy carriers may be more useful and leak more information than lower frequency, high-energy carriers. Also in many situation, such as attacking cryptographic tokens, PDAs and SSL accelerators, the EM side-channel is the only powerful side-channel available since the power side-channel is not accessible.

Next we illustrate that different EM carriers carry different information and leakages via some EM side-channels are different from and incomparable to power side-channel leakage and therefore the EM side-channel can sometimes be more powerful than the power side-channel.

Just like the power side-channel, the EM side-channel signals can be used to perform attacks like simple/differential electromagnetic attacks (SEMA/DEMA) which are the analogues of SPA and DPA. This is because, like power signals, EM emanations are correlated to each active bit in the state of device at an instant in time. Also, by comparing the correlation plots of DEMA/DPA for a particular algorithmic bit using different EM channels as well as the power side-channel, one can compare how a particular bit leaks in the various side-channels. Figures 15.24 and 15.25 show the correlation plots for the correct hypothesis for the DES algorithm running on a smart card using three different EM channels (AM demodulation done at different carrier frequencies) as well as the power side-channel. These correlation plots are aligned in time for all the channels with the power side-channel being the solid line and the different EM channels being different styles of broken lines. These plots show the extent to which the algorithmic bit (an S-box output bit in this case) leaks into different side-channels. Figure 15.24 shows that the bit leaks differently at different times in these channels. Figure 15.25 shows the case where the bit leaks substantially in two of the EM channels, somewhat less prominently in the third EM channel and hardly leaks within the power side-channel. In smart cards, this is a common occurrence for several ALU-oriented instructions since power leakages are biased toward instructions that access memory and consume more energy. We term these instructions "bad instructions", i.e., instructions where information leakage in an EM channel is significantly greater than the corresponding leakage in the power side-channel. In the 6805-based smart card, several bit-test instructions turned out to

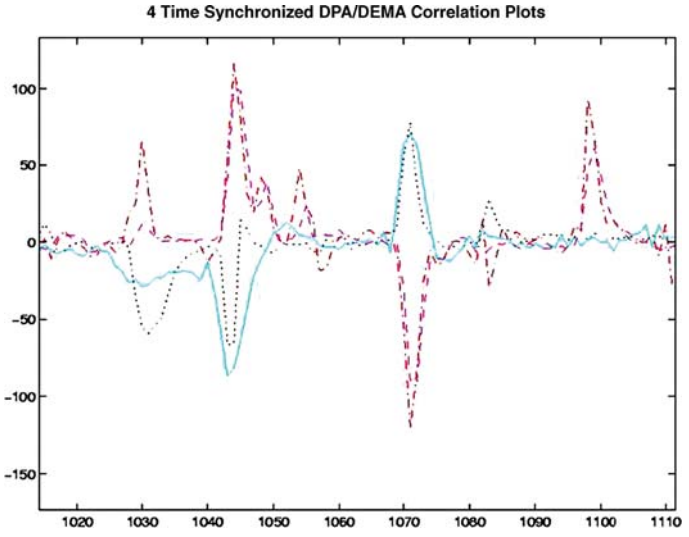

Fig. 15.24 DPA and three DEMA correlation curves (aligned).

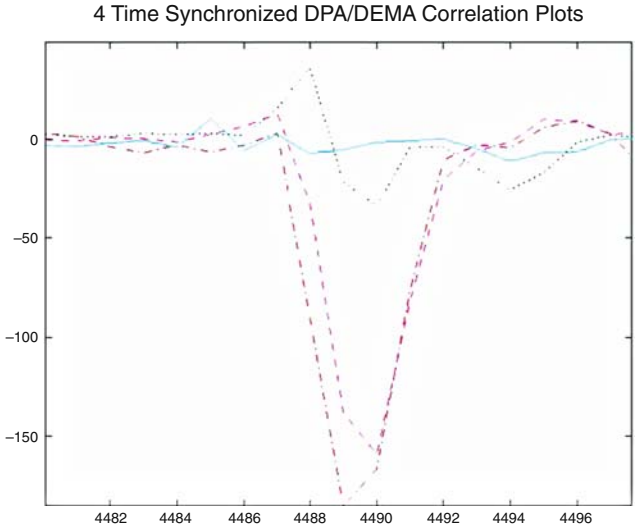

Fig. 15.25 DPA and three DEMA correlation curves (aligned) where the bit leaks substantially.

be bad instructions: the value of the bit being tested leaked into the EM side-channel but not in the power side-channel. Figures 15.26 and 15.27 shows two traces where the tested bit is different and same, respectively, and the highlighted portion of the signal is significantly different in these two cases, thus directly leaking the bit. The power side-channel on the other hand did not carry this information.

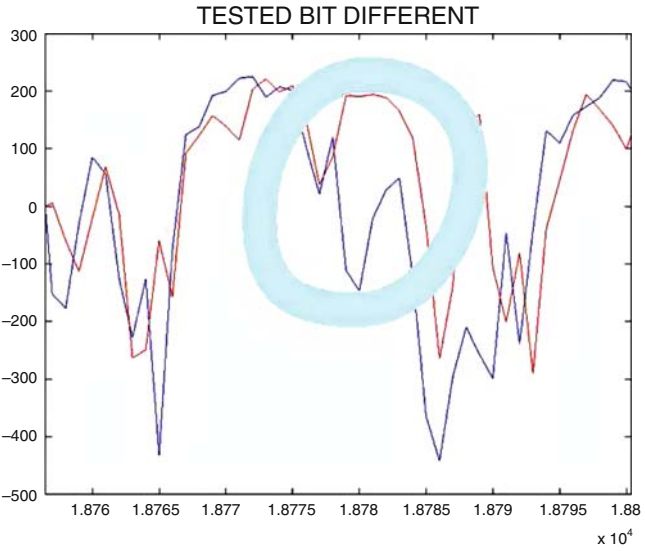

Fig. 15.26 Two EM signals for a bit-test operation: bits different.

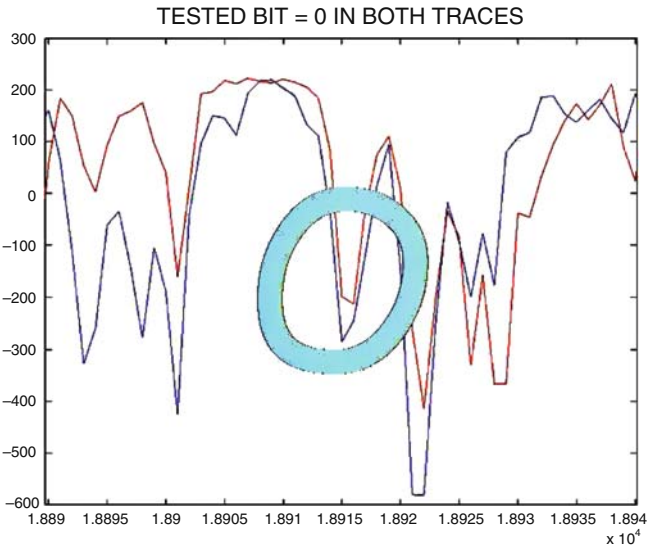

Fig. 15.27 Two EM signals for a bit-test operation: bits same.

15.6 Using EM to Bypass Power Analysis Countermeasures

In general all architectures have bad instructions and typically, in smart cards, these tend to be the ALU-intensive instructions rather than bus-intensive instructions. These bad instructions provide an avenue to break power analysis-resistant implementations.

A common assumption behind many power analysis countermeasures is that once the basic execution sequence is free from simple power analysis (SPA) attacks, there is enough noise/uncertainty in each power trace to prevent direct recovery of sensitive information. Then various techniques such as masking [3, 6] and nonlinear key update [7] can be used to further amplify this uncertainty to prevent the adversary from learning information from multiple samples. If bad instructions are used in a DPA-resistant implementation, then this assumption of limited leakage from a single sample is violated and vulnerabilities get created. For example, if the EM leakage is very large, then the DPA-resistant implementation may be vulnerable to SEMA. If the EM leakage is moderate then higher-order EM attacks on masking DPA countermeasures become possible as was shown in [1].

15.7 Quantifying EM Exposure

In order to assess vulnerability of a device to EM and other side-channels it is imperative that there be an assessment methodology in place to determine the extent of the leakage and the effectiveness of the countermeasures. In the case of EM, this

assessment can be quite complex since there are several possible interception points and at each interception point multiple EM signals are available by considering different carriers and demodulations. One has to consider different adversaries and classes of attacks including low-cost attacks by limited adversaries who can capture only one signal at a time, to more powerful adversaries that can capture multiple signals and perform complex signal processing operations, as well as unbounded adversaries that can capture as many signals as they wish from a bounded set of sensors and perform any feasible processing on these signals.

In some cases a sound methodology to assess EM vulnerabilities in these cases is feasible. This will be covered in the chapter on improved techniques for side-channel analysis.

15.8 Countermeasures

EM analysis countermeasures include circuit redesign to reduce unintentional emanations and techniques to reduce the S/N ratio observed by the adversary. For example, EM shielding and/or the introduction of additional noise can reduce the S/N ratio. Another option is to set up physically secure zones where entry is restricted, to prevent the adversary from capturing a strong EM signal.

A systematic way to minimize EM exposure is outlined in the following quote from the NACSIM 5000 TEMPEST Fundamentals document:

> "The prevention of TEMPEST problems can best be accomplished by being attentive to the problem throughout every stage of the equipment or system design and development. Due to the many ways that information is processed in an equipment, there are many ways that compromising emanations can be generated. It is nearly impossible to completely prevent the generation of such compromising emanations. Therefore, the TEMPEST design objectives should be to (a) keep the amplitude and frequency spectrum of compromising emanations as low as possible (i.e., below the appheable limit); (b) prevent RED signals from coupling from RED to BLACK lines or circuits; and (c) to prevent emanations from escaping from the equipment through electromagnetic or acoustical radiation or through line conduction. When involved in retrofitting non-TEMPEST designed equipments, many of the methods identified herein, in addition to encapsulation techniques, may be useful in meeting design objectives."

However, the following cautionary quote from NACSIM 5000 also outlines why, from a practical perspective, such EM attack resistance is unlikely to be present in most systems.

> "In typical baseband communication or data processing circuit designs, minimum attention is given to suppression of unintentional emanations. Design engineers do not realize the importance of component selection, interconnections, or layout in minimizing signal emanations. Draftspersons, who are unfamiliar with electrical engineering fundamentals, are frequently employed in the design of PC boards and interconnecting leads. Occasionally, this chore is delegated to a computer, which follows a minimal number of rules governing circuit applications and circuit interconnections. As a result, undesired signal emanations will probably be detected when the equipment must be proven TEMPEST hazard-free."

Once the basic EM leakage is minimized to prevent SEMA-style attacks, then other randomization-based countermeasures that have been used in the context of DPA, such as random masking or computing with shares or nonlinear updates of sensitive information, may be used as countermeasures against DEMA attacks.

15.9 Projects

Pre-requisite: A wideband radio and embedded device, e.g., a cellphone or PDA

1. Using a wideband radio how can you determine the clock frequency and harmonics of the processor? Verify by checking the device specifications. What professional equipment can be used to quickly determine the clock signals within the device?
2. *(Advanced: Assuming that You Can Program the PDA.)* Use your knowledge of the processor clock frequency and instruction set to write a program that loops with a frequency of around 1000 Hz or any other frequency in the audible range, till a key/button is pressed. Execute the program on the PDA. Then, slowly scan the parts of the spectrum that are covered by your radio (using AM or FM demodulation). If the processor clock and harmonics are within a band covered by the radio, you should be able to hear the 1000 Hz tone at several different center frequencies. Each of these bands represents a potential EM side-channel that leaks information about the computation occurring within the processor.
3. Now that you have determined the EM bands where there is leakage from the CPU, how would you use this information to set up EM capturing equipment and carry out a SEMA/DEMA attack on the device?
4. *Locating Compromising Emanations from Device Display:* While manipulating the information displayed by the device (e.g., either by running an application that regularly updates the screen or manually updating what is displayed), slowly scan the parts of the spectrum (either AM or FM demodulation) that your radio covers. At several frequencies you should be able to hear audible sounds whenever the screen changes. These are frequencies at which information about the contents of the screen can leak. Actual attacks to capture the screen will depend on the specifics of how the display is being refreshed.
5. *EM Propagation (Advanced):* First conduct experiment in exercise 2 to obtain the 1000 Hz tone indicating EM leakage from the device. Place the device with the running program inside a completely enclosed metal box (or a cardboard box covered with aluminum foil). Can your receiver still capture the 1000 Hz tone outside the metal box? Why not? Now place the device on a metal box that has one or a few small openings (e.g., by creating a small opening within the foil-covered cardboard box). Again, try to obtain the 1000 Hz tone with your receiver. Move the receiver around the box to locate where the signal is strongest. Where is the signal the strongest? Why?

References

1. D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In B. Kaliski, Ç. K. Koç, and C. Paar editors, *Proceedings of CHES 2002*, Lecture Notes in Computer Science, vol. 2523, pp. 29–45, Springer, 2002.
2. A. V. Borovik and C. D. Walter. A Side Channel Attack on Montgomery Multiplication. Private technical report, Datacard platform seven, July 1999.
3. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. Wiener editor, *Proceedings of Advances in Cryptology, CRYPTO '99* Lecture Notes in Computer Science, vol. 1666, pp. 398–412, Springer, 1999.
4. Dynamic R1550. Dynamic Sciences International Inc, R 1550 Receiver. Specifications available at http://www.dynamic-sciences.com/r1550.html.
5. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In Ç. K. Koç, D. Naccache, and C. Paar editors, *Proceedings of CHES 2001*, Lecture Notes in Computer Science, vol. 2162, pp. 251–261, Springer, 2001.
6. L. Goubin and J. Patarin. DES and Differential power analysis (The "Duplication" method). In Ç. K. Koç and C. Paar editors, *Proceedings of CHES 1999*, Lecture Notes in Computer Science, vol. 1717, pp. 158–172. Springer, 1999.
7. P. C. Kocher and J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener editor, *Proceedings of Advances in Cryptology CRYPTO '99*, Lecture Notes in Computer Science, vol. 1666, pp. 388–397, Springer-Verlag, 1999.
8. M. G. Kuhn and R. J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In D. Aucsmith editor, *Information Hiding 1998*, Lecture Notes in Computer Science 1525, pp. 124–143, Springer-Verlag, 1998.
9. J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and countermeasures for smart cards. In *Proceedings of e-Smart 2001*, Lectures Notes in Computer Science (LNCS), vol. 2140, pp. 200–210, Springer, 2001.
10. W. Schindler. A Timing attack against RSA with chinese remainder theorem. In Ç. K. Koç and C. Paar (eds.) *Proceedings of CHES 2000*, Lecture Notes in Computer Science, vol. 1965, pp. 109–124, Springer, 2000.
11. W. van Eck. Electromagnetic radiation from video display units: An evesdropping risk? *Computers & Security*, vol. 4, pp. 269–286, 1985.
12. C. D. Walter and S. Thompson. Distinguishing exponent digits by observing modular subtractions. In D. Naccache editor, *Proceedings of CT-RSA 2001*, Lecture Notes in Computer Science, vol. 2020, pp. 192–207, 2001.

Chapter 16
Leakage from Montgomery Multiplication

Colin D. Walter

16.1 Introduction

Modular multiplication $P = A \times B \bmod M$ is a fundamental operation in most public key cryptography. Its efficiency is usually critical in determining the overall efficiency of a system because it is the main component in modular exponentiation and in elliptic curve point multiplication. There are several algorithms which can be chosen for performing modular multiplication, of which those by Barrett [1], Montgomery [6] and Quisquater [2] are the most widely known. Most optimisations which can be applied to one modular multiplication algorithm can also be applied to the others, so that all have the same overall complexity [9]. However, Montgomery's method is rather more straightforward to implement; generally less work is involved in achieving the optimisations.

This chapter delves into certain aspects of Montgomery's algorithm: it seeks to retain the advantages of simple and efficient code while at the same time addressing the issue of side channel leakage from the final conditional subtraction. We study the main loop and the final conditional subtraction separately in order to determine a fully precise specification for the output and hence determine how much data are leaked through the conditional subtraction side channel. This enables us to fix the leakage very satisfactorily.

16.2 Montgomery Reduction

Modular multiplication is really a combination of two processes: multiplication and modular reduction. These are generally *interleaved* for space efficiency reasons: this keeps the intermediate values within a very small multiple of the modulus. This is also called the *integrated* approach. With the *separated* technique, the multiplication

Comodo CA Ltd,
e-mail: Colin.Walter@comodo.com

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_16,

© Springer Science+Business Media, LLC 2009

is performed completely beforehand, allowing more time-efficient methods to be employed. This is then followed by modular reduction of the product. We start by looking at this process of reduction.

Definition 16.1. Suppose positive integers M and R have no common factor. Then,

i) A' is called the Montgomery reduction of A modulo M with respect to R if $A' \equiv AR^{-1} \bmod M$ and $AR^{-1} \leq A' < AR^{-1}+M$.
ii) Conversely, $\overline{A}$ is called a Montgomery representation or Montgomery residue of A with respect to R if it satisfies $\overline{A} \equiv AR \bmod M$. When R is clear, $\overline{A}$ is called an M-residue.

Note that this definition contains two different inverses of R: one is the residue modulo M which is the modular inverse of R in $\mathbb{Z}/M\mathbb{Z}$, the other is the rational number which is the fractional inverse of R in $\mathbb{Q}$. The context, indicated by the presence or absence of "$\bmod M$", makes the intended choice clear.

Co-primality of M and R guarantees that there is a one-to-one correspondence between a complete set of residues $\bmod M$ and the set of residues $\{R'.R \bmod M \mid 0 \leq R' < M\}$. So there is a value R' such that $R'.R \equiv 1 \bmod M$, i.e., R has an inverse $\bmod M$. So the Montgomery reduction exists when M and R have no common factor, and the bounds ensure that it is unique. For cryptographic applications the co-primeness property usually holds: R is generally a power of 2 so that division by R is easily performed by shifting, whereas M is a product of some large primes, and hence odd.

The Montgomery reduction A' of $A \bmod M$ can be obtained by finding integers A' and Q satisfying

$$A'R - QM = A$$

and such that Q is in the interval $[0\ldots R[$. When R has the form $R = r^n$, the solution can be generated using the same process as in Hensel's lemma, which solves the equation iteratively modulo higher and higher powers of r.

When A and M have representations over base (or radix) r with digits a_i and m_i, respectively, the Henselian process for obtaining the Montgomery reduction A' with respect to $R = r^n$ is given in Figure 16.1. There, ${m_0}^{-1}$ is the inverse of m_0 modulo r. Usually the digits occupy words of memory, so that $r = 2^k$ where k is the number

Function MonRed(A, M, r, n): A'
Pre-condition: M and r are co-prime.
Post-condition: $A' \equiv Ar^{-n} \bmod M$ with $Ar^{-n} \leq A' < Ar^{-n}+M$.

$A' \leftarrow A$
For $i \leftarrow 0$ to $n-1$ do
  $q_i \leftarrow -a'_0 m_0^{-1} \bmod r$
  $A' \leftarrow (A' + q_i M) \operatorname{div} r$
Return A'

Fig. 16.1 Montgomery modular reduction.

of bits per word. Then the division by r in the last line of that figure is simply a shift by one position of an array of words.

The choice of digit q_i in the loop guarantees that the division by r is exact. Thus, if the digits q_i are formed into the "quotient" $Q = \sum_{i=0}^{n-1} q_i r^i$ then the output satisfies $A' = (A+QM)r^{-n}$ where $Q < r^n = R$, as required. The bounds on A' now follow, showing it is the Montgomery reduction of input A.

The term Ar^{-n} becomes smaller as n is increased. Hence, by choosing n such that $A < Mr^n$ the output satisfies $A' < 2M$ and an extra conditional subtraction of M will yield the least non-negative residue of $AR^{-1} \bmod M$. For example, if the input A were obtained as a product of two reduced residues modulo M (i.e., least non-negative) then $A < M^2$ and we would just need n such that $M < r^n$ in order to achieve an output which is also fully reduced by the extra conditional subtraction.

Unlike classical modular reduction, the choice of quotient digit q_i does not depend on the most significant digit of A' but on its least significant. This means that q_i can be determined precisely without waiting for carry propagation to be completed in the previous loop iteration. This is advantageous for application in a systolic array where the processing elements perform digit level computations [10].

16.3 Montgomery Modular Multiplication

Instead of pre-computing the product $A \times B$, the Montgomery reduction of $A \times B$ modulo M can be obtained by interleaving the multiplication and the reduction, as in Figure 16.2. We need one of the inputs, say B, to have a representation to base r with at most n digits.

It is easy to verify the code of Figure 16.2 from its similarity to MonRed and by observing that the non-modular operations compute $A \times B$ with a shift equivalent to a factor of r^{-n}. In fact, taking $B = 1$ yields the MonRed algorithm. As in the MonRed algorithm, there is no upper bound on the value of input A. Moreover, the bound on B is in terms of n and not M. Thus there is no need to ensure the inputs are least non-negative residues modulo M.

Function MonPro(A, B, M, r, n): C
Pre-condition: M and r are co-prime, $B = \sum_{i=0}^{n-1} b_i r^i < r^n$.
Post-condition: $C \equiv ABr^{-n} \bmod M$ and $ABr^{-n} \leq C < M + ABr^{-n}$.

$C \leftarrow 0$
For $i \leftarrow 0$ to $n-1$ do
  $q_i \leftarrow -(c_0 + a_0 b_i) m_0^{-1} \bmod r$
  $C \leftarrow (C + b_i A + q_i M) \operatorname{div} r$
Return C

Fig. 16.2 Montgomery modular multiplication without conditional subtraction.

Since the output of the ith iteration is exactly analogous to the output of the nth, for which $B < r^n$, we know that the partial product C is less than $M+A$ throughout the calculation. This gives a bound on how large the register for C needs to be. For most applications $M < r^n$ and $A < r^n$ so that C requires one more bit than r^n. A further extra word may also be needed for intermediate results before the shift down corresponding to the division by r. A detailed time and space efficiency analysis of different methods to compute the update to the partial product C is given by Koç, Acar and Kaliski [3].

Setting $Q = \sum_{i=0}^{n-1} q_i r^i$ gives $C = (AB+QM)r^{-n}$ and so $Q \approx ABr^{-n}/M$ when C is bounded by a small multiple of M. To be precise, if $C-\delta M$ is the smallest non-negative residue, then $Q+\delta = \lfloor ABr^{-n}/M \rfloor$ yields the integer quotient. So further conditional subtractions of M from C to obtain the least non-negative residue can be combined with incrementing Q to yield the integer quotient.

The normal presentation of the algorithm includes a final conditional subtraction of M to yield an output less than M. It is omitted in this first version of Montgomery multiplication for three reasons: it is unnecessary when MonPro is used for exponentiation, it is a strong source of side channel leakage and, for non-fully reduced inputs, more than one subtraction of M may be necessary. However, there are two useful versions of Montgomery multiplication which include a final conditional subtraction. They are given in Figures 16.3 and 16.4.

It is easy to check that the input bounds imply the output bounds in both cases. Moreover, the bounds are such that outputs can be used as inputs to another execution of the same algorithm. This is very convenient for applications involving exponentiation. The second version, with bound R, is marginally more efficient than the first for two reasons. First, this is because the comparison is easier to implement: typically it just requires looking at an overflow bit rather than performing a poten-

Function $\text{MonPro}^{(M)}(A,B,M,r,n)$: C
Pre-condition: M and r are co-prime, $A < M$, $B < r^n$.
Post-condition: $C \equiv ABr^{-n} \bmod M$ and $C < M$.

$C \leftarrow \text{MonPro}(A,B,M,r,n)$
if $C \geq M$ then $C \leftarrow C - M$

Fig. 16.3 Montgomery modular multiplication with bound M.

Function $\text{MonPro}^{(R)}(A,B,M,r,n)$: C
Pre-condition: M and r co-prime, $A<R$, $B<R$, $M<R$ for $R = r^n$.
Post-condition: $C \equiv ABr^{-n} \bmod M$ and $C < R$.

$C \leftarrow \text{MonPro}(A,B,M,r,n)$
if $C \geq R$ then $C \leftarrow C - M$

Fig. 16.4 Montgomery modular multiplication with bound R.

tially full-length subtraction. Second, the frequency of the subtractions is usually lower for the second version.

16.4 Exponentiation

All exponentiation algorithms consist of a sequence of products. Therefore, in order to use MonPro for modular exponentiation, it suffices to adjust for the extra power of R and to check that the output from one use of MonPro satisfies the bounds on the input required for its use in any subsequent MonPro. If

$$Z = X \times Y \bmod M$$

is one of the modular multiplications during the exponentiation when normal modular products are computed, then the usual choice for the corresponding multiplication using MonPro is

$$\overline{Z} = \text{MonPro}(\overline{X}, \overline{Y}, M, r, n)$$

which operates on the corresponding M-residues. Comparing powers of R, we find $\text{MonPro}(\overline{X}, \overline{Y}, M, r, n) \equiv \overline{X} \times \overline{Y} \times R^{-1} \equiv XR \times YR \times R^{-1} \equiv X \times Y \times R \equiv Z \times R \equiv \overline{Z} \bmod M$. Thus, the entire exponentiation is done correctly with MonPro on the corresponding M-residues if the input is adjusted to an M-residue and the output is re-adjusted back from an M-residue. An example of this is given in Figure 16.5 for the square-and-multiply method of exponentiation.[1]

To ensure the correct power of R in all operands, the evaluation of $S = T^N \bmod M$ requires a pre-processing step to convert T to its M-residue:

Function $\text{MonExp}^{(R)}(T, N, M, r, n, R^{(2)}) : S$
Pre-conditions: M and r co-prime, $T < M < R$ for $R = r^n$, $R^{(2)} < R$,
$R^{(2)} \equiv R^2 \bmod M$, and $N = (n_{k-1} \ldots n_2 n_1 n_0)_2$ with $n_{k-1} = 1$.
Post-condition: $S = T^N \bmod M$ and $0 \leq S < M$.

$\overline{T} = \text{MonPro}^{(R)}(TR^{(2)}, M, r, n)$
$\overline{S} \leftarrow \overline{T}$
For $i \leftarrow k-2$ downto 0 do
    $\overline{S} \leftarrow \text{MonPro}^{(R)}(\overline{S}, \overline{S}, M, r, n)$
    If $n_i = 1$ then $\overline{S} \leftarrow \text{MonPro}^{(R)}(\overline{S}, \overline{T}, M, r, n)$
$\overline{S} \leftarrow \text{MonPro}^{(R)}(\overline{S}, 1, M, r, n)$

Fig. 16.5 Square-and-multiply exponentiation with $\text{MonPro}^{(R)}$.

[1] Strictly speaking, M must be square-free to avoid the possibility of output $S = M$, which is forbidden in the post-condition of the code. The output bound is treated later in this section.

$$\overline{T} = \text{MonPro}(T, R^2 \bmod M, M, r, n)$$

and the post-processing step to convert back from an M-residue:

$$S = \text{MonPro}(\overline{S}, 1, M, r, n).$$

It is easy to see that the first of these introduces a factor of R, while the second removes such a factor. Corresponding similar pre- and post-processing steps also correctly introduce and eliminate a factor of R when $\text{MonPro}^{(M)}$ or $\text{MonPro}^{(R)}$ is used throughout the exponentiation instead of MonPro – simply replace MonPro by $\text{MonPro}^{(M)}$ or $\text{MonPro}^{(R)}$ as appropriate.

For the large exponents N typical of public key cryptography, these two extra modular multiplications have a negligible cost when compared with the advantages of a simpler multiplication algorithm. However, $R^{(2)} \equiv R^2 \bmod M$ also needs to be pre-computed and stored. Since M is usually fixed for many exponentiations, the time penalty for this can normally be amortized over the lifetime of the modulus.

The other main requirement for using Montgomery modular multiplication in exponentiation is that the output from one multiplication satisfies the pre-conditions for inputs to the subsequent multiplications. This is plainly the case if $\text{MonPro}^{(R)}$ is used throughout: inputs and outputs are both bounded by R. To use $\text{MonPro}^{(M)}$ throughout, the condition $M < R$ must be added in order to guarantee that the "B" input is small enough. Then all inputs and outputs are bounded by M.

Now suppose MonPro is used in the exponentiation algorithm. In order to achieve a common bound, say B, on all inputs and outputs of MonPro, it is necessary that $M+B^2R^{-1} \le B$. This just requires that the quadratic $B^2R^{-1}-B+M$ has real roots, i.e., $4M < R$. This is achieved by choosing a large enough value for R, that is, a sufficiently large n where $R = r^n$. Then any bound B can be chosen as long as it is between the two roots. This is readily seen to be the case for bounds such as $B = 2M$ or $B = \frac{1}{2}R$, or even a weighted average $B = 2\lambda M + \frac{1-\lambda}{2}R$ where $0 \le \lambda \le 1$ [13].

For all three modular multiplication algorithms, the above bounds are sensible conditions on cryptographic inputs T to an exponentiation. With bound $B = M$ or $B = R$ for $\text{MonPro}^{(B)}$, the initial conditions $T < B$ and $R^{(2)} < R$ ensure that $\overline{T} < B$, making $\overline{T}$ suitable for subsequent use in the exponentiation. Thereafter, every input to $\text{MonPro}^{(B)}$ is bounded above by B, so that its output is also bound by B. For MonPro, the initial conditions $T < M$ and $R^{(2)} < R$ ensure that $\overline{T} < B$ for the acceptable bound $B = 2M$. Thus, the pre-processing input $R^{(2)}$ need not be fully reduced in any of three cases; R is an adequate bound for it in each case.

For $\text{MonPro}^{(B)}$ with bound $B = M$ or R, and for MonPro with bound $B = 2M$ or $\frac{1}{2}R$, the loop of the post-processing modular multiplication by 1 generates output S satisfying $S < M+BR^{-1} < M+1$. Hence, for none of the three algorithms does a final subtraction take place nor is it necessary to obtain a fully reduced output, except possibly when $S = M$ occurs.

However, a loop output of $S = M$ is almost impossible. For $\text{MonPro}^{(M)}$, this is irreconcilable with the obvious properties $0 < \overline{S} < M$ and $\overline{S} \equiv 0 \bmod M$. When M is square free, $S \equiv 0 \bmod M$ implies $T \equiv 0 \bmod M$. In this case, a pre-condition

of $0 \leq T < M$ forces $T = 0$. Then the output of every modular multiplication is 0, ensuring that $S = 0$, so that $S = M$ does not occur. So exponentiation with MonPro never uses even a single subtraction to achieve a fully reduced output. Otherwise, with MonPro$^{(R)}$ or MonPro in the non-square-free case, a single subtraction may be required to obtain a fully reduced output [11].[2]

16.5 Space and Time Comparisons

In this section, the time and space requirements of exponentiation methods using the three modular multiplication algorithms are compared: the standard MonPro$^{(B)}$ with conditional subtraction and bound $B = M$ or R, and MonPro with no subtraction and bound $B = 2M$ or $\frac{1}{2}R$.

The radix of the number representations here is $r = 2^k$ where k is the native bit length of words in the processor which performs the modular multiplication. Typical word lengths are small powers of 2, such as 8-, 16-, 32- and 64-bit. Standard key lengths for RSA normally coincide with multiples of these, such as 1024, 1536 and 2048. The same is true for many of the standard prime fields $\mathbb{F}_p$ used in elliptic curve cryptography [8]. Consequently, to achieve the property $M < R$ required for MonPro$^{(B)}$ with minimal cost, the property $R < 2M$ frequently holds as well, that is, R is the smallest power of 2 greater than M. So, discarding the final subtraction and using MonPro for exponentiation instead of MonPro$^{(B)}$ comes at an initial cost of increasing the number of iterations n, probably by just 1, to ensure $4M < R = r^n$.

So, with standard key and word lengths and the minimal choice for n, the register containing C in MonPro$^{(B)}$ needs to have one more bit or one more word than M because loop output values can be up to $M+B$ in magnitude where $B = M$ or R. Increasing n by 1 to make $4M < R$ with an input bound of $B = 2M$ or $\frac{1}{2}R$ generates intermediate values also less than $M+B$, but this is still below R. So MonPro exponentiation needs *no* extra words for the intermediate calculations. Indeed, it requires only *two* more bits for C than for M. So the loops in the modular multiplications of MonPro and MonPro$^{(B)}$ exponentiations have the same computational space requirements in a fully word-based implementation, and MonPro uses only one more register bit when the top word is reduced to contain only the bit positions that are needed.

Because both manipulate the same number of words, there is also unlikely to be any time difference between single loop iterations in MonPro and MonPro$^{(B)}$. The topmost incomplete word or equivalent individual bits cannot be processed any faster than full words because the clock speed is set to that of the slowest word operation.

So the main time and space differences will be between the final conditional subtraction when MonPro$^{(B)}$ is used for exponentiation and the extra loop iteration when MonPro is used. A leak-resistant implementation of MonPro$^{(B)}$

[2] See Exercise 2 for the non-square-free case.

exponentiation will always perform the subtraction, but needs an extra register to hold the pre-subtraction value of C (the "minuend") as well as its post-subtraction value. On the other hand, being more complex than a subtraction, the extra iteration of MonPro may require more time. Taking into account their relative complexity, the loop iteration is most likely to take the same time as a subtraction or double that time since the clock frequency will probably be chosen to make word-level multiply accumulate and subtraction operations take the same time. At a theoretical level, the subtraction itself might be equivalent to about half an extra loop iteration, and selecting the result of the subtraction or its minuend equivalent to another half of a loop iteration. This would make MonPro and MonPro$^{(B)}$ take essentially the same time.

The subtractor itself may require significant extra dedicated hardware and associated data manipulation may require extra time. Moreover, code size will be increased by having to incorporate instructions for the subtraction. As observed in Section 16.4, MonPro exponentiation requires no final subtraction, so that the extra hardware and code may not be necessary, although it is likely that other cryptographic operations on the chip will require them.

Although the precise cost will be implementation specific, this discussion indicates that using MonPro$^{(B)}$ with its conditional subtraction will be more expensive in hardware than using MonPro for exponentiation, and the time requirements are essentially identical. In conclusion, exponentiation using MonPro with $4M < R$ and no final subtraction is a cost-effective and straightforward solution to the problem of side channel leakage from conditional subtractions.

16.6 Side Channel Analysis

A substantial embarrassment to many smart card manufacturers in the 1990s was the public discovery that naïve implementations of Montgomery's algorithm can cause substantial side channel leakage, enabling private keys to be recovered from fewer than a 100 uses of the key [4, 12]. The main problem arises from the conditional subtraction in MonPro$^{(M)}$ and MonPro$^{(R)}$ which, because of the length of keys, takes a large number of clock cycles to complete. It is therefore very evident in any EMR or power trace.

Nowadays there are many effective counter-measures to prevent such leakage, not least of which is using MonPro for exponentiation rather than MonPro$^{(M)}$ or MonPro$^{(R)}$. Another counter-measure is to modify MonPro$^{(M)}$ and MonPro$^{(R)}$ slightly so that the subtraction is always performed. Then the original loop output value C is kept, and the new or old value is selected according to the sign of the new value. In this way the leakage is considerably reduced. Other counter-measures are discussed in other chapters; here we limit ourselves to measuring the leakage from MonPro$^{(M)}$ and MonPro$^{(R)}$ and varying the parameter R in order to minimise it.

Assume that an implementation of exponentiation using a public modulus and private exponent is under attack and that all conditional subtraction events can be

observed clearly through a side channel. First, we make the simple observation that when the conditional subtraction occurs in one modular multiplication but not in another, then the multiplications must involve different arguments. Suppose an attacker can choose the input to an exponentiation with the secret key on the target device and he can observe individual conditional subtractions. With knowledge of the public modulus and exponentiation algorithm, he can also write a software simulation of the exponentiation which will generate the same sequence of conditional subtractions when it has the same input and uses a correct guess at the secret key. He then guesses the bits of the key in the order that they are consumed by the algorithm. Whenever there is a difference between the conditional subtractions in the side channel leakage and his simulation, he knows the operands at that point differ between the two exponentiations. So he has guessed incorrectly and he backtracks to change his most recent guesses; several previous bits may need to be adjusted. Providing the subtractions occur with probability not too close to 0 or 1, he has a good chance of recovering the whole of the private key in this way.[3] Such an attack uses leakage from a single exponentiation and can be applied to both RSA and ECC, as well as other exponentiation-based protocols. The obvious counter-measure is to blind the input text T before exponentiating.

Second, in elliptic curve cryptography, the classical formulae for point addition and point doubling are so different that it is easy to distinguish them in side channel traces. Then, with an algorithm such as square-and-multiply, it becomes trivial to read the pattern of adds and doubles and deduce the secret key. One counter-measure is to use "unified" formulae for both doubling and adding, so that the same sequence of field operations is performed in both cases. Unfortunately, for a point doubling some pairs of these operations have identical arguments, whereas they are different for point additions. When $\text{MonPro}^{(M)}$ or $\text{MonPro}^{(R)}$ is used to implement the associated modular arithmetic, a difference in the behaviour of the conditional subtractions indicates a point addition unequivocally, whereas identical behaviour makes a point doubling more likely to be the case. With the shorter keys which occur in ECC, the attacker is left with a small search space of possible keys which it is often computationally feasible to traverse [15]. This attack does not require the opponent to be able to input data of his own choosing to the exponentiation although it may require repeated use of the key until a sufficiently favourable exponentiation occurs.

Third, a number of side channel attacks depend on recording the frequency of the conditional subtraction at different points in an exponentiation algorithm over a number of executions of it with different data and the same unblinded (secret) exponent. In particular, as we see in the next section, the frequency is different for squarings and multiplications. Hence repeated use of the square-and-multiply algorithm with $\text{MonPro}^{(R)}$ (Figure 16.5) or $\text{MonPro}^{(M)}$, the same secret exponent and random input data T, would enable the sequence of squarings and multiplications to be deduced, and hence the bits of the secret key obtained.

[3] In the next section we find that the probability of a subtraction is at most $\frac{1}{2}$ and, by increasing R, the probability can be made as close to 0 as desired.

From this summary of attacks, it is clear that no implementation of Montgomery modular multiplication or reduction should be allowed a final conditional subtraction which takes execution time which is not constant. The only exception is if the subtraction is extremely rare – a case that may actually arise below for certain choices of the parameters. Otherwise MonPro$^{(M)}$ and MonPro$^{(R)}$ should always perform their subtraction and choose the original or updated value as appropriate.

16.7 Frequencies of Conditional Subtractions

In this section we consider a set $\mathscr{S}$ of executions of MonPro$^{(M)}$ with common modulus M. The aim is to estimate the expected frequency of the conditional subtraction for the main ways in which the set might have arisen. Then side channel leakage about the conditional subtractions can be used to deduce which of the causes is the most likely for $\mathscr{S}$. The set might represent corresponding operations in a number of different exponentiations using the same unblinded key, and the aim may be to determine if these operations were all multiplications or all squarings. Similar results can be obtained for MonPro$^{(R)}$, but they are considerably complicated by having non-uniform distributions for the inputs and outputs: inputs and outputs have ranges greater than M, so that some residue classes modulo M have more than one representative.

As an example, consider the set of all MonPro$^{(M)}$ multiplications $A\times B \mod M$ for M=7 and R=8. Since $R^{-1} \equiv 1 \bmod M$, the output C satisfies $C \equiv AB \bmod M$. So, with ABR^{-1} as a lower bound, the outputs from the MonPro loop are those given in Table 16.1. The overall frequency of the conditional subtraction is $\frac{5}{49}$. However, restricting to squares $A\times A \bmod M$, the frequency of the subtraction becomes $\frac{1}{7}$, which is a little greater. If the input A is fixed to A=5, then the probability of the subtraction for random B rises to $\frac{2}{7}$. Different fixed values of A result in other quite different probabilities.

There are three main types of set $\mathscr{S}$ to consider: those arising from squaring, those arising from multiplications in which both arguments are free to assume any values independently and those arising from multiplications in which one argument

Table 16.1 A modular multiplication table: unreduced outputs from MonPro(A,B) when $M = 7, R = 8$. The five bold entries require reduction.

0	0	0	0	0	0	0
0	1	2	3	4	5	6
0	2	4	6	1	3	5
0	3	6	2	5	**8**	4
0	4	1	5	2	6	3
0	5	3	**8**	6	4	**9**
0	6	5	4	3	**9**	**8**

has the same value for the whole set. There are other sets of interest which are not discussed. For example, none of the above matches the set of all multiplications from a single m-ary or sliding windows left-to-right exponentiation (unless $m = 2$): because one input, say A, is taken from a set of only $\log_2 m$ distinct multipliers, it is neither constant nor uniformly distributed.

Several reasonable, simplifying assumptions are made in order to derive the frequencies of the subtraction for $\mathscr{S}$. They are often very hard to justify theoretically, but several are closely related to the diffusion properties on which the associated cryptography relies. First,

- it is assumed that $\phi(M) \approx M$.

The "Euler phi" function having a value close to M means that M is a product of a small number of large, not necessarily distinct, primes, as in the case of RSA and $\mathbb{F}_p$. The property $\phi(M) \approx M$ just states that almost every number is prime to M.

Suppose input A of MonPro is prime to M. Then A has an inverse modulo M. Therefore, if the other argument B has a uniform distribution modulo M, so will the output C. So at least one input being uniformly distributed means that, to a very good approximation, the output is also uniformly distributed. Such uniformity is propagated from input to output through every instance of MonPro in an exponentiation if the initial input is uniformly distributed. Due to formatting and construction restrictions, the inputs to the exponentiation may not be uniform in practice, but diffusion occurs so rapidly during exponentiation that, except possibly for the initial one or two multiplicative operations, the uniformity can be assumed.

Let A, B and Z be discrete random variables over the interval of integers $0\ldots M-1$ corresponding respectively to the two MonPro$^{(M)}$ inputs and the variation in output C of the MonPro loop within the interval $[ABr^{-n}, M+ABr^{-n}[$. Suppose

- A, B and Z are all independent and uniformly distributed

and let π_{mu} be the probability that the final subtraction takes place. Then $\pi_{mu} = pr(Z+ABR^{-1} \geq M) = \frac{1}{M^3}\sum_{Z=0}^{M-1}\sum_{A=0}^{M-1}\sum_{B=0}^{M-1}(Z+ABR^{-1} \geq M) \approx \frac{1}{M^3}\sum_{A=0}^{M-1}\sum_{B=0}^{M-1} ABR^{-1} \approx \frac{1}{M^2}\int_0^M \int_0^M ABR^{-1} dA\, dB$ where $Z+ABR^{-1} \geq M$ is 0 or 1 according to the truth of the inequality. (The approximations arise from using real numbers instead of integers and are very accurate for cryptographic-sized moduli.) So

$$\pi_{mu} \approx \frac{1}{4} MR^{-1} \tag{16.1}$$

This is the probability of the subtraction for a set $\mathscr{S}$ of multiplications under the above hypotheses. Suppose the ith operation in a set of exponentiations is always a multiplication and that the inputs to the exponentiations are uniformly distributed modulo M. Then π_{mu} is the probability of a subtraction when $\mathscr{S}$ is the corresponding set of instances of MonPro$^{(M)}$.

Now let π_{sq} be the probability that the final subtraction takes place when MonPro$^{(M)}$ is used to square a uniformly distributed input. For the same definitions as above, suppose

- A and Z are independent and uniformly distributed.

Then $\pi_{sq} = pr(Z+A^2R^{-1} \geq M) = \frac{1}{M^2}\sum_{Z=0}^{M-1}\sum_{A=0}^{M-1}(Z+A^2R^{-1} \geq M) \approx \frac{1}{M^2}\sum_{A=0}^{M-1} A^2R^{-1} \approx \frac{1}{M^2}\int_0^M A^2R^{-1}dA$, whence

$$\pi_{sq} \approx \frac{1}{3}MR^{-1} \tag{16.2}$$

Unlike sets of multiplications, in practice most sets of squarings satisfy the criteria to apply this value for π_{sq}. Since $\pi_{mu} < \pi_{sq}$, the subtraction is less frequent for multiplications than for squarings, as in the example with M=7.

Now suppose the value of A is fixed but the argument B is uniformly distributed on $0 \ldots M-1$. Let π_A be the probability that the conditional subtraction takes place. For definitions as before, assume also that

- B and Z are independent and uniformly distributed.

Then $\pi_A = pr(Z+ABR^{-1} \geq M) = \frac{1}{M^2}\sum_{Z=0}^{M-1}\sum_{B=0}^{M-1}(Z+ABR^{-1} \geq M) \approx \frac{1}{M^2}\sum_{B=0}^{M-1} ABR^{-1}$ $\approx \frac{1}{M^2}\int_0^M ABR^{-1}dB$. Hence

$$\pi_A = \frac{1}{2}AR^{-1} \tag{16.3}$$

So, for fixed multiplier A, the frequency of subtractions depends strongly on its size. For large A, such as 5 and 6 in the example with $M = 7$, the frequency is highest. At the other extreme, note that the value is correct even for $A = 0$, although it is not prime to M and so causes the output not to be uniformly distributed. As expected, the average value of π_A is π_{mu} when A is uniformly distributed.

16.8 Variance in Frequencies and SCA Errors

If the frequency of the conditional subtraction is used to determine whether the set $\mathscr{S}$ consists of multiplications or squarings, then the accuracy of the decision is important to know.

Let $\mathscr{S}_i$ be the set of ith modular multiplications in a collection of t square-and-multiply exponentiations using the same 1024-bit key. So $|\mathscr{S}_i| = t$. There are about 1500 sets $\mathscr{S}_i$ to classify in order to recover the bit pattern of the secret exponent. If e errors are made, then, with a simple-minded approach in which every bit might be one that has been misclassified, approximately 1500^e different alternatives might have to be tried before the correct exponent is discovered. Of course, operations for sets $\mathscr{S}_i$ with frequencies close to the average of π_{mu} and π_{sq} are the most likely to be mis-classified, and the search should begin there. This would find the correct key much more quickly. Nevertheless, it is clear that the error count e has to be kept very small for this to become the foundation of a computationally feasible attack.

Assuming the inputs to the t exponentiations are independent, the conditional subtractions should occur independently within each $\mathscr{S}_i$. Then the behaviour is that of a binomial random variable for t trials with probability $p = \pi_{mu}$ or π_{sq}. The

expected number of subtractions is tp and its variance is $\sigma^2 = tp(p-1)$. To make use of tables for the normal distribution, it is more likely that we will prefer to work with the probability of the subtraction rather than its total count. In this case, the mean is p and the variance is

$$\sigma_{mu}^2 = \frac{1}{4t}MR^{-1}(1-\frac{1}{4}MR^{-1})$$

$$\sigma_{sq}^2 = \frac{1}{3t}MR^{-1}(1-\frac{1}{3}MR^{-1})$$

or

$$\sigma_A^2 = \frac{1}{2t}AR^{-1}(1-\frac{1}{2}AR^{-1})$$

The point at which classification as a squaring or multiplication is equally likely is the weighted average

$$\pi = \frac{\pi_{sq}\sigma_{mu} + \pi_{mu}\sigma_{sq}}{\sigma_{mu} + \sigma_{sq}} \tag{16.4}$$

If the conditional subtraction occurs less often than this in $\mathscr{S}_i$ then the set probably contains multiplications; if it is greater then the set probably contains squarings. The probability of making an incorrect decision can be obtained by looking up tables for the normal distribution, which approximates the binomial distribution when t is large. It uses the fact that if X is a normal random variable with mean μ and variance σ^2, then $\sigma^{-1}(X-\mu)$ has the N(0,1) distribution, which is tabulated [7]. So, the probability of the count being on the wrong side of π is easily seen to be approximately $Pr(Z > \frac{\pi_{sq}-\pi_{mu}}{\sigma_{mu}+\sigma_{sq}})$, where Z is an N(0,1) random variable. This is the average probability of mis-classifying an operation, but clearly the operation is much more likely to be correctly classified when its subtraction probability is well away from π than when it is close to π.

This error probability is roughly $Pr(Z > \frac{\sqrt{t}}{13.2})$ for a typical value of $MR^{-1} \approx \frac{3}{4}$. Then $t > 1800$ would lead to less than 1 error in the 1500 or so operations for a 1024-bit exponentiation. If R is doubled, the value of interest in the N(0,1) tables is divided by about $\sqrt{2}$. Then around 18 errors appear. More refined attacks on the same data can deduce the key with many fewer exponentiations [14]. However, as the example shows, a very suitable counter-measure is to reduce MR^{-1} by increasing R, because this reduces the incidence of conditional subtractions and hence reduces the side channel leakage.

16.9 A Surprising Improvement

The formulae for π_{mu} and π_{sq} show that one of the easiest ways to reduce leakage is to reduce MR^{-1}. Since the bit length of M is generally fixed, this means increasing the number of iterations n in the Montgomery modular multiplication algorithm MonPro (Figure 16.2). This reduces the frequency of conditional

subtractions. However, as noted in the last paragraph of Section 16.4, MonPro can be used for exponentiation on its own without the final conditional subtraction once $MR^{-1} < \frac{1}{4}$. This is because the extra shifting down then makes the output small enough for re-use as an input to the next occurrence of MonPro.

A straightforward solution to side channel leakage would be to reduce standard key lengths by 2 bits. Then $4M < R$ would hold automatically for the minimum choice of n in a word-based implementation. Consequently, MonPro could be used throughout an exponentiation without the need for extra iterations or conditional subtractions, and all intermediate values would remain within the word boundaries. However, suppose that M reaches the top of the word boundary, so that MonPro requires one more iteration than MonPro$^{(M)}$ or MonPro$^{(R)}$. We will now look in more detail at how this affects the bounds on intermediate values when MonPro is used.

So, assume also that the word size is at least 2 bits and that n is chosen minimally to ensure $4M < R$. This means $r \geq 4$ and $rM < R < 2rM$. Let R' denote the Montgomery multiplier which the standard version MonPro$^{(M)}$ or MonPro$^{(R)}$ would have used. So $R = rR'$ and $M < R' < 2M$. This standard version produces loop outputs bounded by $ABR'^{-1}+M$, which is only just larger than R'. But here MonPro performs an extra division by r. Consequently, the upper bound $ABR^{-1}+M$ on the output is less, making it more likely to satisfy $ABR^{-1}+M < R'$. An interesting question is therefore whether all inputs and outputs would be bounded by R' when the extra iteration is performed, because the hardware requirements would then be lower. Substituting in the bound R', the desired property holds if $R'r^{-1}+M < R'$, i.e., if $M < R'(1-r^{-1})$. Writing this as $M < R'r^{-1}(r-1)$ shows the condition is just that the top word of M is not $r-1$, i.e., not entirely 1s.

Theorem 16.1. ([13], Thm 5.) *Suppose the top digit of M is not $r-1$, that $r \geq 4$ and that MonPro is executed with n iterations where $M < R' = r^{n-1}$, i.e., one more iteration than normal. Then inputs bounded by R' generate outputs bounded by R'.*

In other words, in almost all cases M is such that a single extra iteration of the MonPro loop avoids any need for a conditional subtraction. In effect, it is equivalent to using MonPro$^{(R')}$ but without the conditional subtraction side channel. Intermediate values of C at the end of a loop iteration in MonPro are bounded above by $A+R'$ so that, as usual, one extra register bit is needed.

If one is prepared to exclude the rare cases of inconvenient moduli M with top digit $r-1$, this means that exponentiation can be carried out successfully using MonPro with the usual register sizes and omitting conditional subtractions, providing only that an extra iteration is done in the MonPro loop. Of course, at the end of the exponentiation there is also no need for a conditional subtraction: as noted before, the adjusting MonPro multiplication by 1 reduces the output to less than M.

With the extra iteration, the exceptional cases can be included simply by restoring the conditional subtraction, i.e., by using MonPro$^{(M)}$ or MonPro$^{(R')}$ with an incremented value of n. Then, as in Section 16.7, Equation (16.3), the output can be assumed to be uniformly distributed modulo M, so the probability of exceeding M or R' is at most $\frac{1}{2}r^{-1}$. With a typical key lifetime of 10000 uses, and the difficulty

of capturing that number of side channel traces, we just require 16-bit or larger words for the conditional subtraction to have negligible probability of happening even once per key bit during the lifetime of the key. This effectively eliminates the side channel leakage since there is insufficient data at the information theoretic level to reconstruct the key. To summarise, MonPro$^{(B)}$ (B=M or R') can be used securely for exponentiation with all keys if an extra iteration is performed and a 16-bit or larger processor is used.

16.10 Conclusions

This chapter has carefully derived input and output bounds for several versions of Montgomery modular multiplication in order for it to be of use in exponentiation. Some pre- and post-processing is necessary, and the pre-computation of a multiplier $R^{(2)}$ whose one-time cost can be amortised over the lifetime of the modulus M.

This analysis enabled the frequencies of conditional subtractions to be deduced accurately, showing that there could be considerable side channel leakage from the ability to observe the subtractions.

It was shown how this side channel could be closed by increasing the number of iterations in the MonPro algorithm. At the further cost of one extra register bit, or the exclusion of some extreme modulus values, the conditional subtraction can be totally eliminated. However, even if the conditional subtraction were retained to include the exceptional cases, the extra iteration reduces the side channel to an unusable level except for these extreme moduli on hardware with a very small word size.

The chapter therefore provides several choices for implementing Montgomery modular multiplication in a manner which effectively eliminates any side channel leakage emanating from the final, conditional subtraction.

16.11 Exercises

1. a. Find 16^{-1} mod 11. Using the output bounds given in Figure 16.2, compute a complete table of output from MonPro when $M = 11$, $R = 16$ and A and B are least non-negative residues modulo M.
   b. Use this table to determine the probability of the conditional subtraction in MonPro$^{(M)}$ for (a) multiplications and (b) squarings which have uniformly distributed inputs. (c) For each value of A calculate the probability of the subtraction for multiplications by fixed A. (d) Compare these probabilities with the theoretical ones derived in Section 16.7.
   c. Repeat (b) with MonPro$^{(R)}$ in place of MonPro$^{(M)}$ (including part (d)).
   d. Compute a complete table of output from MonPro when $M = 7$, $R = 32$ and A and B are strictly bounded above by $2M$. What is the largest value in the

table? For each value $B = M, M+1, ..., 2M-1$, if both inputs are strictly less than B, find out how many outputs are B or larger.

2. a. Let $M = 9$, $T = 3$, $N = 2$, $R = 16$ in the computation of $S = T^N \bmod M$ using MonPro$^{(R)}$. Find $R^{-1} \bmod M$ and determine the two possible values for $R^{(2)}$. Deduce the values of $\overline{T}$ and $\overline{S}$. Hence show that the output S still requires a final subtraction to be fully reduced. Why does this not contradict the claim in the text that final subtractions are unnecessary?
   b. Repeat (a) for the same inputs but with MonPro and $R = 64$.
   c. Prove that using MonPro or MonPro$^{(R)}$ to compute $S = 3^N \bmod 9$ will always give output $S = 9$ when $N \geq 2$ and R satisfies the usual constraints.
   d. Suppose P is a prime such that P^2 divides M, $T = M/P$ and $N \geq 2$. With the usual requirements on R, prove that using MonPro or MonPro$^{(R)}$ to compute $S = T^N \bmod M$ will always give output $S = M$.
3. Let N = DDMMYY be today's date (a pseudo-random exponent), M the first prime greater than the number of this page, R the first number greater than $\frac{9}{8}M$ which is prime to M and chosen input text $T = \lfloor \frac{1}{4}M \rfloor$.

   a. Use a calculator to compute a value for $R^{(2)}$ and hence calculate the pre-processing value $\overline{T}$ given by MonPro$^{(M)}$ for an exponentiation.
   b. Convert N into binary and hence list the operations required to perform a left-to-right square-and-multiply exponentiation with exponent N. Use this list to create a list of the associated values of $\overline{S}$ when MonPro$^{(M)}$ is used to evaluate $T^N \bmod M$. Include in the list information about whether or not the conditional subtraction occurs.
   c. Now, using the values for $\overline{T}$, R and M, reconstruct all the possible exponents which generate the same list of the conditional subtractions as N. You should create a binary tree of options for the bits and traverse the tree systematically, calculating the corresponding values of $\overline{S}$ and checking to see whether or not the behaviour of the conditional subtraction is the same. When the behaviour of the subtraction is different, the branch can be pruned as it cannot represent the true value of N.
   d. Count the number of nodes in the tree which have been visited. Build an argument to justify that this number is linear in the length of the exponent. Is such a search computationally feasible for a large exponent? What would happen to this count if (a) R/M were larger? or (b) T/M were larger?

16.12 Projects

1. Choose any modulus M with no factors less than 2^8, say, such that $(M-1)^2 \bmod M$ can be computed correctly and easily in the machine arithmetic that is available to you. Select a Montgomery constant $R > M$. R need not be a power of 2 for this exercise, but must be prime to M.

a. Write a program to perform exponentiation with 16-bit exponents using $\text{MonPro}^{\langle M \rangle}$.
b. Check that your code performs correctly by comparing results with a classical implementation of modular exponentiation.
c. Check also that no final subtraction is required to get output less than M.
d. Modify your code to collect data about the occurrences of the conditional subtraction in the ith modular multiplication for each i. (Clearly $1 \leq i < 32$.)
e. Generate $t = 10^3$ random exponentiations using the same modulus and same exponent and collect the conditional subtraction frequencies for each position i in the exponentiation.
f. Verify that the frequencies match those expected from Section 16.7, at least on average.
g. Compute the means and standard deviations of the frequencies for (a) the multiplications and (b) the squarings. Hence compute the value of π given in Equation (16.4).
h. Use your value for π to partition the operations into multiplications and squarings. How many operations are mis-classified in this way? If the number of multiplications is known and used as the partitioning point, how many are mis-classified then?
i. Repeat (e)–(h) several times for different numbers of exponentiations to see how the number of mis-classified operations varies with t.

2. a. Repeat Project 1 with $\text{MonPro}^{\langle R \rangle}$ in place of $\text{MonPro}^{\langle M \rangle}$, this time selecting R such that $(R-1)^2 \bmod M$ is easily computable on your available machine.
   b. Divide the interval $[0, R]$ into, say, 50 or 100 sub-intervals and modify the exponentiation code to collect the frequencies for the output of $\text{MonPro}^{\langle R \rangle}$ lying in each sub-interval. Plot these frequencies on a graph for several different values of M such that $\frac{1}{2}R < M < R$. Explain why the curve representing these frequencies rises from 0 to $R-M$, is horizontal between $R-M$ and M and then falls from M to R.
   c. Modify your data collection in (ii) to separate the sub-interval frequencies for multiplication outputs from those of squaring outputs. Are the graphs of these frequencies different? If so, attempt to explain the difference.
3. Collect subtraction frequencies from the exponentiation software from Projects 1 or 2 when the same modulus and exponent are used for a set of $t = 10^3$ exponentiations.

   a. Use the length of the exponent (16 here) and the total number of modular multiplications in an exponentiation to deduce the number m of multiplications and the number s of squarings.
   b. Let $\mathscr{M}$ be the set consisting of the m operations with the lowest frequencies for the conditional subtraction, and let $\mathscr{S}$ be the complementary set consisting of the other s operations. As before, count how many operations are mis-classified if those in $\mathscr{M}$ are assumed to be multiplications and those in $\mathscr{S}$ are assumed to be squarings.

c. Modify your software to collect the data from the ith operation as a t-dimensional vector over $\{0,1\}$ where 0 indicates that no subtraction took place and 1 that a subtraction did take place. Call the vector v_i. For each operation, compute the average Hamming distance between v_i and the vectors in $\mathscr{M}$. Does this distance depend on whether the operation is a multiplication or a squaring?
d. Suppose the m operations for which the distance of part (iii) is smallest represent multiplications. Re-allocate all the operations in $\mathscr{M}$ and $\mathscr{S}$ under this assumption. Does this decrease the number of errors?
e. Repeat the re-allocation process of part (d) a number of times and observe whether the number of errors decreases or increases with each iteration of the process.
f. Repeat a similar re-allocation process on the initial partition of part (b) using the average Hamming distance of v_i from the vectors in set $\mathscr{S}$.
g. Inconsistencies between the results of (v) and (vi) must represent allocation errors in one or the other case. However, are there any operations which are still incorrectly allocated? If not, reduce the value of t to see when errors start appearing. If so, increase t to see if the number of errors decreases. Also, how does the number of inconsistencies vary with t?
h. Repeat all the above for exponents with a variety of larger and larger lengths. Does the proportion of errors change as the length increases when t is fixed? How does the total number of errors vary for fixed t as the length varies?

4. Following on from Exercise 2, investigate more thoroughly the cases where the computation of $S = T^N \bmod M$ using MonPro$^{(M)}$ might require a final conditional subtraction in the post-processing phase. Look first at when M is a prime power, then at more general non-square-free M.

References

1. P. D. Barrett. *Implementing the Rivest Shamir Adleman public key encryption algorithm on standard digital signal processor*, Advances in Cryptology – CRYPTO '86, pp. 311–323, Springer, 1987.
2. J.-J. Quisquater. *Presentation at the rump session of Eurocrypt '90*.
3. Ç. K. Koç, T. Acar, and B. S. Kaliski, Jr. *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, 16(3): 26–33, 1996.
4. P. Kocher. *Timing Attack on Implementations of Diffie-Hellman, RSA, DSS, and other systems*, Advances in Cryptology – CRYPTO '96, N. Koblitz (editor), LNCS 1109, pp. 104–113, Springer-Verlag, 1996.
5. P. Kocher, J. Jaffe, and B. Jun, *Differential Power Analysis*, Advances in Cryptology – CRYPTO '99, M. Wiener (ed.), LNCS **1666**, pp. 388–397, Springer-Verlag, 1999.
6. P. L. Montgomery. *Modular Multiplication without Trial Division*, Mathematics of Computation, 44(170): 519–521, 1985.

7. NIST/SEMATECH. *Cumulative Distribution Function of the Standard Normal Distribution* §1.3.6.7.1 in the "e-Handbook of Statistical Methods" at http://www.itl.nist.gov/div898/handbook/, 2006.
8. NIST. *Digital Signature Standard*, Appendix 6 (July 1999), Federal Information Processing Standard (FIPS) 186-2, Jan 2000.
9. S. E. Eldridge and C. D. Walter. *Hardware Implementation of Montgomery's Modular Multiplication Algorithm*, IEEE Trans. Comp. 42: 693–699, 1993.
10. C. D. Walter. *Systolic Modular Multiplication*, IEEE Trans. Comp. **42**, 1993, 376–378.
11. C. D. Walter. *Montgomery Exponentiation Needs No Final Subtractions*, Electronics Letters, 35(21): 1831–1832, October 1999.
12. C. D. Walter and S. Thompson. *Distinguishing Exponent Digits by Observing Modular Subtractions*, Topics in Cryptology – CT-RSA 2001, D. Naccache (editor), LNCS 2020, pp. 192–207, Springer-Verlag, 2001.
13. C. D. Walter. *Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli*, Proceedings of CT-RSA 2002, LNCS 2271, pp. 30–39, Springer-Verlag, 2002.
14. C. D. Walter. *Longer Keys may facilitate Side Channel Attacks*, Selected Areas in Cryptography – SAC 2003, LNCS 3006, pp. 42–57, Springer-Verlag, 2004.
15. C. D. Walter. *Simple Power Analysis of Unified Code for ECC Double and Add* Proceedings of CHES 2004, LNCS 3156, pp. 191–204, Springer-Verlag, 2002.

Chapter 17
Randomized Exponentiation Algorithms

Colin D. Walter

17.1 Introduction

A *randomized algorithm* for function f takes the usual inputs for f together with a stream of random numbers and combines them in a way such that partial or complete knowledge of the atomic operations used to compute f does not easily reveal the values of some or all inputs. The output of f is still computed correctly, but the value is independent of the random input stream.

In this chapter we consider randomized algorithms for the exponentiation function and assume that side channel leakage reveals a certain level of partial knowledge about the arithmetic and read/write operations performed on the manipulated big numbers. Our object is to make it computationally infeasible for an attacker to use this information to deduce the secret exponent during its use over the lifetime of a cryptographic token.

For example, in the usual square-and-multiply algorithm (Figure 17.1, [6]), full knowledge of the sequence of squares and multiplies immediately determines the complete exponent uniquely. Specifically, there is an exponent bit for every square; and every time the square is followed by a multiplication the bit must be a 1, whereas it must be a 0 when the square is followed by another square.

As a rule, leaked information is rarely without error; a number of squares may be incorrectly recorded as multiplications and *vice versa*. Hence there is normally some error correction to be performed. If the number of errors is small enough, a search of nearby keys will discover the true value D in a computationally feasible time. Its correctness can be confirmed easily by using the corresponding public key E and the relation $P^{ED} = P$. Traversing the search space must often be done intelligently, selecting the most probable alternatives first in order to have any hope of finding the key.

In typical protocols using RSA [13], the same secret key D is re-used a number of times, during which it may or may not be possible to blind it by, for example, adding

Comodo CA Ltd
e-mail: Colin.Walter@comodo.com

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_17,

© Springer Science+Business Media, LLC 2009

Inputs: M, bit representation of $D = d_{n-1}d_{n-2}\ldots d_2 d_1 d_0$
Output: $C = M^D$

$C \leftarrow 1$
For $i \leftarrow n-1$ downto 0 do
Begin
    $C \leftarrow C^2$
    If $d_i \neq 0$ then $C \leftarrow C * M$
End

Fig. 17.1 Square-and-Multiply Exponentiation.

a random multiple of $\phi(N)$ where N is the public modulus. Data about the atomic operations can be accumulated over repeated use of the secret key and might be combined successfully to reveal the key. However, many protocols, such as ECDSA [2], generate a fresh random number on each occasion for use as the secret key. In this case there is only one chance to obtain the key and, moreover, it may be practically impossible to extract useful side channel information from the cryptographic token at the same time as using it to perform a real signature. Thus there is a variety of contexts for which protection against leakage is desirable. Different randomized algorithms are appropriate for these, and they have differing costs in terms of execution time and space, and code size, as well as demands on the supply of random numbers.

The body of this chapter describes the main randomized algorithms for exponentiation, provides an overview of possible attacks on them under likely leakage models, and considers how the inevitable errors affect results. One corollary of the definition of a randomized algorithm is that successive uses of the same inputs will result in different sequences of operations. Consequently, Kocher's averaging over many uses of the same key will prove useless [7]. However, we motivate the discussion further by showing how leaked data might be combined in Kocher-like fashion to extract operator and operand information from a *single* use of a secret key. This contrasts with attack details in earlier chapters where such information was only obtained after averaging over many applications of the same key. We also conclude with the important but counter-intuitive result that use of longer keys may actually *weaken* a cryptosystem rather than strengthen it when side channel leakage occurs.

17.2 The Big Mac Attack

This attack [16] applies to m-ary exponentiation (Figure 17.2, [6]) and to all similar algorithms which use a table of pre-computed digit powers of the input ciphertext C. Big Mac is so-called because the digits d_i of the secret key are determined individually and independently, just like the flavours tomato, beef burger, lettuce, cheese, etc. of different layers in a certain well-known fast food product which is too large to be consumed in any other way.

Inputs: M, base m representation of $D = d_{n-1}d_{n-2}\ldots d_2 d_1 d_0$
Output: $C = M^D$

Pre-computation of the table:
$M^{(1)} \leftarrow M$
For $i \leftarrow 2$ to $m-1$ do $M^{(i)} \leftarrow M * M^{(i-1)}$

Exponentiation of the message:
$C \leftarrow 1$
For $i \leftarrow n-1$ downto 0 do
Begin
    $C \leftarrow C^m$
    If $d_i \neq 0$ then $C \leftarrow C * M^{(d_i)}$
End

Fig. 17.2 m-Ary exponentiation.

Using the notation of Figure 17.2 in which the exponent has a representation in base m, the attacker first has to distinguish the processes $C \leftarrow C^m$ of raising to the mth power and $C \leftarrow C * M^{(d_i)}$ of multiplication by the digit power of M. He must then partition the multiplications into disjoint sets for which the digits d_i have the same values. The method for doing this is intimated below. Normally m will be a power of 2 so that raising to the mth power is a sequence of $\log_2 m$ squarings. For convenience, we will assume that the mth power can be detected by recognizing squares from multiplies. Once the partitioning has been performed, there are $(m-1)!$ ways of associating specific different digit values with the $m-1$ sets of multiplications. One of these choices will yield the sought key. In fact, the pre-computations can be used to determine the map from sets of multiplications to digits.

The background to the attack is the fact that the power consumed by a hardware multiplier depends to some extent on the Hamming weight of the inputs [21]. This means that if we were to look at the averaged power trace for a large number of word-by-word multiplications $a \times b$ where a varies randomly and b is fixed then we would obtain a result which is, at least to some extent, characteristic of b. For standard classical multipliers, averaged traces for words b of equal Hamming weight will be closer together than those for words of different weights. However, all we need for the attack to succeed is for the power trace to vary measurably between enough groups of word values. This depends mostly on the experimental technique and the trouble and expense to which attacker and designer are prepared to go. With a bit of experimentation, one can associate a probability of b having a particular value for a given average trace.

The idea behind the attack is to apply the usual averaging process of Kocher's power analysis [7] to digit $\times$ big integer multiplication traces rather than to exponentiation traces. Kocher takes a number of exponentiation traces associated with the same secret exponent and arranges them so that parts corresponding to the same exponent digit are aligned. He then takes an average to improve the signal-to-noise ratio. Here we cut the trace of $C \leftarrow C \times M^{(d_i)}$ into sub-traces corresponding to constituent operations $C \leftarrow shift(C) + c_j \times M^{(d_i)}$ where c_j is a word-level digit

of C, i.e., a group of consecutive bits of C used as a single input to the hardware multiplier of the cryptographic token. These sub-traces (one for each j) are aligned and averaged to give a trace which should be characteristic of $M^{(d_i)}$. The dependency on the words c_j of C is averaged away. This is done for all n digits of the key D. The n averaged traces are compared using the Euclidean distance between them. It turns out that traces for which the digits have the same value are close together whereas those for different valued digits are noticeably further apart. This enables the partitioning to be performed.

Typically in RSA the inputs C and M will have around 1024 bits and an 8- or 16-bit hardware multiplier will be used. This means that C will be broken into about 64 words, and this is the number of traces which are averaged. Unless C has a really exceptional value (such as 0), this is enough to remove any dependency on C in the averaged trace. Moreover, as $M^{(d_i)}$ also has about 64 words, it is very unlikely that a pair of them will share enough digits of similar characteristics to be confused when their averaged traces are compared.

The same technique is applied to distinguish the squarings (or multiplications) used in obtaining the mth power of C. In this case the averaged trace is that of a random operand C' rather than that of $M^{(i)}$ for some digit value i. This trace is not close to that of any of the multiplications involving $M^{(d_i)}$ nor to any of the operands used in the other mth power computations. Hence, when the partitioning process is applied to all multiplications in the exponentiation, including those in the mth powers, we can identify the mth power computations and group together the sets of multiplications for which the same exponent digit has been used. So, unless the key lengths are small and the multipliers are large, one might expect the digits d_i, and hence the key D, to be recovered more or less accurately from a single exponentiation.

17.3 Digit Representation and Exponentiation Algorithms

All of the randomized exponentiation algorithms in this chapter depend on a randomized recoding of the binary representation of the secret key D. This is done by one of the change-of-base algorithms in Figures 17.3 and 17.4, the latter being a more complex version of the former in which the base m can be randomly varied instead of being fixed. In both figures the function mod$'$ includes a random input which allows a limited number of alternative output digits d_i subject to the property that the division $(D-d_i)/m$, resp. $(D-d_i)/m_i$, in the next line is exact. So the outputs satisfy

$$D=\sum_{i=0}^{n-1} d_i m^i = ((\dots(d_{n-1}m+d_{n-2})m+\cdots+d_2)m+d_1)m+d_0 \tag{17.1}$$

and

$$D=((\dots(d_{n-1}m_{n-2}+d_{n-2})m_{n-3}+\cdots+d_2)m_1+d_1)m_0+d_0 \tag{17.2}$$

respectively.

Inputs: $D \geq 0$, base $m > 1$
Outputs: n, base m representation of $D = (d_{n-1} \ldots d_2 d_1 d_0)_m$
$i \leftarrow 0$
While $D > 0$ do
Begin
    $d_i \leftarrow D \bmod' m$
    $D \leftarrow (D - d_i)/m$
    $i \leftarrow i + 1$
End
$n \leftarrow i$

Fig. 17.3 Change-of-base algorithm for fixed base.

Input: $D \geq 0$
Outputs: n, base sequence $m_{n-1} \ldots m_2 m_1 m_0$, digit sequence $d_{n-1} \ldots d_2 d_1 d_0$
$i \leftarrow 0$
While $D > 0$ do
Begin
    Select base m_i
    $d_i \leftarrow D \bmod' m_i$
    $D \leftarrow (D - d_i)/m_i$
    $i \leftarrow i + 1$
End
$n \leftarrow i$

Fig. 17.4 Variable base representation algorithm.

Figure 17.3 just provides the usual, standard representation to some (fixed) base m when mod$'$ is taken to be the usual mod function which returns the least non-negative remainder on division by m. So $m = 2$ will give the binary representation, and $m = 10$ the decimal version of D. With this fixed choice for mod$'$, successive executions of the algorithm will always give the same representation. It yields the normal m-ary exponentiation algorithm when the digits are fed into the exponentiation algorithm given in Figure 17.5.

A well-known example of the application of Figure 17.4 is in the sliding windows exponentiation algorithm. As before, take mod$'$ to be the mod function. The base m_i is chosen to be $m = 2^r$ if the remaining, unrecoded part of D is odd, and to be 2 otherwise. This gives windows of r or 1 bits. The digits for the r-bit windows are all odd and those for the 1-bit windows are all 0. When these are fed into Figure 17.5, the sliding windows exponentiation algorithm is obtained.

Inputs: M, representation of $D = d_{n-1} \ldots d_2 d_1 d_0$
with respect to bases $m_{n-1} \ldots m_2 m_1 m_0$
Output: $C = M^D$

Pre-computation: a table containing $M^{(d)} = M^d$ for each digit value d
$C \leftarrow 1$
For $i \leftarrow n-1$ downto 0 do
Begin
$C \leftarrow C^{m_i}$
If $d_i \neq 0$ then $C \leftarrow C * M^{(d_i)}$
End

Fig. 17.5 Left-to-right exponentiation.

The m-ary and sliding windows algorithms process the bits or digits of the exponent D from left to right, i.e., from most to least significant. However, the square-and-multiply algorithm for exponentiation can also process the digits in the opposite order, as in Figure 17.6. The cost difference between the two directions for base 2 is almost entirely context specific, depending on, for example, how one moves data around in registers.

However, for larger m, the left-to-right and right-to-left versions of the exponentiation algorithm allow one to trade memory requirements for execution time. The left-to-right version requires space for the pre-computed powers of M but the right-to-left version has to spend time computing the digit powers of M (which has a new value) at each loop iteration.

For cryptographic purposes it is usually desirable to recode the exponent in the same order as the digits are consumed in the exponentiation. That means using the change-of-base algorithms here with the right-to-left exponentiation algorithm. The high point of this chapter is the MIST algorithm which does things in this order. If the opposite order is desired and D is stored in binary, then the new

Inputs: M, representation $D = d_{n-1} \ldots d_2 d_1 d_0$
with respect to bases $m_{n-1} \ldots m_2 m_1 m_0$
Output: $C = M^D$

$C \leftarrow 1$
For $i \leftarrow 0$ to $n-1$ do
Begin
If $d_i \neq 0$ then $C \leftarrow C * M^{d_i}$
$M \leftarrow M^{m_i}$
End

Fig. 17.6 Right-to-left exponentiation.

Input: Binary representation of $D = b_{z-1} \dots b_2 b_1 b_0 \geq 0$
Outputs: n, base sequence $m_0 m_1 m_2 \dots m_{n-1}$, digit sequence $d_0 d_1 d_2 \dots d_{n-1}$

$i \leftarrow 0$
carry $\leftarrow 0$
While $z > 0$ do
Begin
    borrow $\leftarrow$ carry
    Choose window size $r_i \leq z$
    if $r_i = z$ then carry $\leftarrow 0$ else choose carry
    $m_i = 2^{r_i}$
    $d_i \leftarrow$ borrow$\times m_i + (b_{z-1} \dots b_{z-r_i})_2 -$ carry
    $i \leftarrow i+1$
    $z \leftarrow z - r_i$
End
$n \leftarrow i$

Fig. 17.7 Variable base recoding algorithm.

bases must be powers of 2, and the change of base is achieved by recoding groups of bits from left to right, as in Figure 17.7. Of course, this makes raising to the power m particularly easy, but the subscripts are inevitably reversed from normal terminology, giving $D = ((\dots(d_0 m_1 + d_1)m_2 + \dots + d_{n-3})m_{n-2} + d_{n-2})m_{n-1} + d_{n-1}$. As we see shortly, this is used very imaginatively in Itoh's overlapping windows method [4].

17.4 Liardet–Smart

For elliptic curves, Liardet and Smart [8] suggested using the variable base recoding of Figure 17.4 where the base selection is a randomly chosen power $m_i = 2^{r_i}$ of 2 bounded above by $r_i \leq R$. This choice is detailed in Figure 17.8 where Random(R) returns a randomly chosen integer in the interval [1,R]. The function mod$'$ of Figure 17.5 is deterministic, being the (signed) residue of least absolute value (taking 1 when D is odd and $r_i = 2$).

The recoding is used in left-to-right exponentiation applied to perform point multiplication in an elliptic curve context. So the terminology becomes "additions" and "doublings" instead of "multiplications" and "squarings". Then the precomputed table of Figure 17.5 need only contain the odd multiples of the input point up to 2^{R-1}: negative digits are dealt with by a point subtraction of the corresponding positive multiple of the input point. So the space efficiency is that of a sliding window with $(R-1)$-bit windows for which all digit multiples have to be computed.

If Random(R) were to have a uniform distribution over $[1,R]$ then the average window size for the odd digits would be $(R+1)/2$. So the time efficiency of the algorithm would be close to that of the equivalent sliding windows algorithm whose window size is $(R+1)/2$. Of course, the distribution of bases could be biased to favour larger values in order to increase execution efficiency.

As the windows now occupy arbitrary positions in the addition/doubling sequence, there will be both adds and doubles in any given position of the side channel traces if the same key is re-used. This should make it virtually impossible to deduce meaningful information from averaging a number of traces. Moreover, if the pattern of adds and doubles can be determined for a single use of the key, there is still the problem of identifying which digit occurs. Even more difficult is to distinguish the sign of the occurring digit because the same operands are used for both signs.

This is an excellent algorithm for protocols such as ECDSA [2] where the secret key is used just once, provided there is not too much side channel leakage. There is an exercise at the end of the chapter to determine the computational cost of key recovery under an expected leakage model. If the classical algorithms for point addition and point doubling are used then the different numbers and types of the constituent field operations could lead to a very accurate determination of the sequence of adds and doubles. So balanced code [1] may be advisable in combination with this algorithm.

Inputs: D, R
Output: m_i

If $(D \bmod 2) = 0$ then $m_i \leftarrow 2$
else
Begin
    $r \leftarrow$ Random(R)
    $m_i \leftarrow 2^r$
End

Fig. 17.8 Liardet–Smart base selection.

Table 17.1 Some recodings of $13 = 1101_2$ with $R = 3$ and their add/double traces.

4	3	2	1	0	4	3	2	1	0
	1	1	0	1		DA	DA	D	DA
1	–	$\bar{1}$	0	1	DA	D	DA	D	DA
–	–	3	0	1	D	D	DA	D	DA
	1	1	–	1		DA	DA	D	DA
1	–	$\bar{1}$	–	1	DA	D	DA	D	DA
–	–	3	–	1	D	D	DA	D	DA
1	0	–	–	$\bar{3}$	DA	D	D	D	DA

17.4.1 Attacking the Algorithm

If the secret key is re-used unblinded, and the pattern of adds (A) and doubles (D) is leaked with few errors, then the situation is less happy. An example is given in Table 17.1 where possible digit sequences on the left are spaced out to indicate the intervening doubling operations, and the corresponding operation sequences, referred to as "traces", are given on the right. So $- \; - \; \bar{3}$ indicates base 2^3 with digit -3 and the corresponding sequence of adds and doubles is $D\,D\,DA$. By pairing each A with a "D", corresponding Ds are aligned with their associated bit position given at the head of each column. For uniformity, there is an initial $\ldots DA$ for the leading non-zero digit, although efficient code would omit it.

In order to determine the value of bits at position i or just above, we will ignore the parts of the traces to the right of position i. For simplicity, assume this part is deleted, i.e., the i rightmost occurrences of D are removed, and any As therein. Next, partition the trace segments into two sets, Tr_i^A and Tr_i^D, according to whether their rightmost operation (in position i) is A or D. Assume there are enough traces to show all the possible patterns of adds and doubles around this position. If only D occurs (i.e., Tr_i^A is empty), then the ith bit must be 0 since the representation using only base 2 would generate A if the bit were 1. The same argument applies if only A occurs. So the bits of index 0 and 1 must be 1 and 0, respectively, in the example of Table 17.1.

Write D_i for the value of the input binary key D from the most significant bit down to, and including, bit i. Then the traces in Tr_i^A represent the value D_i or $D_i + 1$ according to whether the next (i.e., less-significant) digit is non-negative or not. Clearly, as digit d_i is odd for these traces, it must be the odd one of the values D_i or $D_i + 1$ which is represented. So the bit pattern in the number represented by the traces Tr_i^A is identical to that in D_i with the sole possible exception of position i. Assuming there are enough traces for base 2 to have been chosen at position i, we will have A at position $i+1$ if, and only if, the bit is 1 at that position. This can be seen in Table 17.1 where we can use this to deduce the values of bits in positions 1 and 3. In fact, we can have no more As until the next bit which is 1. So we can deduce that bit 2 is 1 from the trace set Tr_0^A. As the first trace only goes up to position 3, we know the input has at most 4 bits, all of which have now been determined.

An attacker may only be able to make, say, 10 measurements with enough accuracy to deduce the patterns of adds and doubles. Consequently, a few bits may be undetermined [19]. (If the arguments are performed carefully, none should actually be incorrect.) However, it then requires surprisingly little computational power to deduce the key.

On the other hand, if adds can scarcely be distinguished from doubles, life is much more difficult for the attacker. His main problem in performing this attack is to align the doubles. Without the ability to do this, the sub-traces corresponding to given key bits cannot be aligned. Their random movement within side channel traces seems to average away useful information except at the key ends. He could select the very longest traces to guarantee only base 2 was used and average them

to deduce their common pattern, but, of course, there will be no such traces because the probability of generating them is too small for cryptographic-sized keys. So, overall, any uncertainty over interpreting the side channel leakage seems to increase dramatically the difficulty of extracting the key.

In conclusion, this algorithm should only be used after careful regard to its context. In particular, it should not be employed where the same key is used repeatedly without blinding unless side channel leakage is low.

17.5 Oswald–Aigner Exponentiation

Another randomized algorithm was proposed by Oswald and Aigner [11]. For ease of presentation, the description here is slightly modified. The base is fixed at $m = 2$ and the digit set is $\{-1,0,1,2\}$. In the digit recoding phase (Figure 17.3), the randomization occurs in the digit selection function mod′ which chooses –1 or 1 when D is odd and 0 or 2 when D is even. However, choice is only possible in certain cases: 2 is only allowed when the previous non-zero digit was -1, namely when a "Carry" has been propagated to obtain the next value of D in the recoding; and -1 is only allowed when there is no such Carry being propagated. Termination is forced by selecting 1 if $D = 1$ and 2 if $D = 2$. This is described in detail in Figure 17.9, and some recodings of 29 are given on the left side of Table 17.2.

Input: $D \geq 0$
Outputs: n, and representation of $D = (d_{n-1} \ldots d_2 d_1 d_0)_2$
$i \leftarrow 0$
Carry $\leftarrow$ False
While $D > 0$ do
Begin
    If $D = 1$ then $d_i \leftarrow 1$ else
    If $D = 2$ then $d_i \leftarrow 2$ else
    If Carry then
    Begin
        If $(D \bmod 2) = 1$ then $d_i \leftarrow 1$ else $d_i \leftarrow 0$ or 2
        If $d_i \neq 0$ then Carry $\leftarrow$ False
    End
    else
    Begin
        If $(D \bmod 2) = 0$ then $d_i \leftarrow 0$ else $d_i \leftarrow 1$ or -1
        If $d_i = -1$ then Carry $\leftarrow$ True
    End
    $D \leftarrow (D - d_i)/2$
    $i \leftarrow i + 1$
End
$n \leftarrow i$

Fig. 17.9 Oswald–Aigner digit generation.

For exponentiation, the right-to-left method of Figure 17.6 is preferred because the digits are then consumed in the same order as they are generated. In an elliptic curve context, the digit -1 causes no problems as point subtraction is as easy as point addition. The digit 2 is processed by re-ordering the loop iteration to perform the point addition (with digit $d_i = 1$) after the point doubling instead of before it. Hence the space efficiency matches that of the equivalent square-and-multiply algorithm. On average half the recoded digits are odd (+1 or –1) and half are even (0 or 2). More precisely, the digits $\{-1, 0, 1, 2\}$ occur in the ratio $\frac{1}{8} : \frac{3}{8} : \frac{3}{8} : \frac{1}{8}$. So the average time efficiency is a little poorer than square-and-multiply because occurrences of digit 2 are more expensive than those of digit 0.

17.5.1 Attacking the Algorithm

As in the attack on the Liardet–Smart algorithm, suppose that adds (A) can be distinguished from doubles (D) reliably in each execution of the exponentiation procedure and that the same key is used many times unblinded as the exponent.

Again, the behaviour of the algorithm at any point depends on the local bit pattern in the binary representation of the key. This bit pattern is reflected in a restricted set of patterns over $\{A, D\}$. From these, the bit pattern can be deduced. For example, the pattern $DAAD$ only arises from the recoding 12 or $\bar{1}2$ (more significant digit on the left). This means a corresponding bit pattern 111 must occur in the binary representation: the middle 1 is needed to generate the digit 2 using a Carry which can only come from the bit on its right being 1, and a 1 is needed on its left to give the recoded digit 1 or $\bar{1}$. This occurs for some traces representing the top three bits in Table 17.2.

Now suppose there are enough traces to generate every possible pattern of operations near a given bit position. From the previous paragraph, we will know every occurrence of 111. Also, the bit pair 00 always causes two doublings with no intervening addition, but for every other bit pair an intervening add is possible. So we can identify every occurrence of 00. Thus 00 cannot occur in the example of Table 17.2. Furthermore, the bit pattern 10 always has one A between the Ds of its two bit positions whereas every other bit pattern allows the Ds to be adjacent for

Table 17.2 Recodings of $29 = 11101_2$ and their traces, both generated right to left.

4	3	2	1	0	4	3	2	1	0
1	1	1	0	1	DA	DA	DA	D	DA
2	$\bar{1}$	1	0	1	AD	DA	DA	D	DA
1	2	$\bar{1}$	0	1	DA	AD	DA	D	DA
2	0	$\bar{1}$	0	1	AD	D	DA	D	DA
1	1	1	1	$\bar{1}$	DA	DA	DA	DA	DA
2	$\bar{1}$	1	1	$\bar{1}$	AD	DA	DA	DA	DA
1	2	$\bar{1}$	1	$\bar{1}$	DA	AD	DA	DA	DA
2	0	$\bar{1}$	1	$\bar{1}$	AD	D	DA	DA	DA

some recoding. Hence all occurrences of 10 will be determined. This shows that 10 must occur over positions 2,1 in the example of Table 17.2. These two cases enable every bit 0 to be determined as well as every bit immediately to the left of a 0. Of course, the remaining undetermined bits must all be 1s, otherwise they would have been determined as belonging to a pattern 00 or 10. In fact, if some traces contain one or more As and others contain no As between two neighbouring Ds, then the corresponding bit pair must be $*1$ for some bit $*$. So every bit 1 can be determined that way. This is the case for the example of Table 17.2 and it reveals the whole key.

However, the attacker may have too few traces to be sure of his deductions about the bits. In this case he looks at the ratios of the number of traces with zero, one or two As between the Ds of a bit pair. Most occurrences of 1 will be determined unequivocally as above, including the majority of occurrences of 111. Otherwise, it is possible to use the operation pattern to assign a probability to the value of each bit pair. For example, no intervening As will make 00 the most likely bit pair, and with a probability that increases with the number of traces available for inspection. As each bit belongs to two pairs (except at the ends), almost all bits are determined with high or complete accuracy. Indeed, with as few as 10 correct traces and a standard key length for elliptic curve cryptography, it is computationally possible to determine any unknown bits and reveal the secret key ([20], Thm. 1).

Greater accuracy is obtained from looking at patterns corresponding to sequences of three or more bits instead of just two and this might overcome problems arising from errors in the traces.

However, the above analysis depends critically on precise alignment of all occurrences of doubles in the traces. With balanced code for adds and doubles [1], this may be difficult because the adds and doubles cannot be distinguished so easily. In fact, it is not clear how to align the traces satisfactorily even if the bits of the key are known as far as the point of interest in the traces. Inexact alignment seems to average away any useful data about the bits except at the ends.

As with the Liardet–Smart algorithm, the security of the Oswald–Aigner method relies on the key being different on each use, or for it to be very difficult to use side channel leakage to distinguish adds from doubles reliably.

17.6 Ha–Moon

There are two randomizing algorithms due to Ha and Moon et al. [3, 25], both presented as left-to-right exponentiation methods.

The first [3] has fixed base $m = 2$ and simply employs the most general binary signed digit (BSD) coding in the change-of-base algorithm (Figure 17.3): it selects digit $d = 0$ when $D \equiv 0 \bmod 2$ and randomly chooses between $d = \pm 1$ when $D \equiv 1 \bmod 2$. When D is odd, the random choice makes the next value of D odd or even with equal probability, and so the occurrence or otherwise of a multiplication does not indicate the value of the next bit in the original input value of D. However, in exactly the same way as with the Oswald–Aigner method, the pattern of additions

and multiplications reveals the exponent with feasible computation when it is re-used about 10 times [10].

The second, improved version by S.-M. Yen et al. [25] uses any fixed 2-power radix and employs digit recoding from most to least significant, so that conversion can be done on-the-fly during a left-to-right exponentiation. An example with base 4 is given in Figure 17.10. It is readily verified that the digit d_i is always in the range 1–14.

Input: Base 4 representation of $D = (b_{n-1} \ldots b_2 b_1 b_0)_4$, $b_{n-1} > 0$
Output: Base 4 representation of $D = (d_{n-1} \ldots d_2 d_1 d_0)_4 + \delta$

Carry $\leftarrow b_{n-1}$
$i \leftarrow n-1$
While $i > 0$ do
Begin
    Borrow $\leftarrow 4*$Carry
    Carry $\leftarrow$ Random from $1, 2, 3$
    $d_{i-1} \leftarrow$ Borrow $-$ Carry $+ b_{i-1}$
    $i \leftarrow i-1$
End
$\delta \leftarrow$ Carry

Fig. 17.10 A Yen–Chen–Ha–Moon digit recoding with base 4.

Regarding time efficiency, the method is similar to m-ary exponentiation. (Here with $m = 4$.) It has the same number of squarings. However, it also has the same number of multiplications as squarings because all the digits are now non-zero, whereas m-ary exponentiation has only $\frac{m}{m-1} = \frac{3}{4}$ of this number. The pre-computations also add marginally to the time, as does the extra digit δ. The space requirement is close to that of m^2-ary exponentiation since the pre-computed table contains $m^2 - 2 = 14$ values. As the digits are non-negative, the technique can be used for modular exponentiation as well as for point multiplication on elliptic curves.

17.6.1 Attacking the Algorithm

The non-zero property of the digits ensures that the pattern of squares and multiplications is always the same and there are no dummy operations to which to apply the safe error attack [24]. The attacks mounted on Liardet–Smart and Oswald–Aigner are therefore impossible here.

However, there are features of the recoded digit values which might be used to extract the bits of the key. Park and Lee [14] observed that the average value of the digit d_{i-1} has 2 for *Carry* and 8 for *Borrow*, making an average of $6 + b_{i-1}$. Hence, minimal leakage of the value of the recoded digit from enough traces will be sufficient to determine b_{i-1} correctly with high probability. For example, the leakage can be turned into probable digit values using the Big Mac attack [16] which was

described in Section 17.2. Now the identical pattern of operations for every exponentiation is in the attacker's favour: he can easily align operations at position $i-1$ and so pool any weak leakage in order to find the average $6+b_{i-1}$. This enables him to recover the secret key D with very few errors. He just needs to collect more trace data to add into his averages if the signal-to-noise ratio is not giving enough correct digits. Consequently, any re-use of a key with such recoding should be combined with random blinding of it. Then the used value of b_{i-1} varies randomly and its average value contains no information.

Thus the second Ha–Moon algorithm exhibits different strengths and weaknesses from those of the Liardet–Smart and Oswald–Aigner algorithms. This may make it more suitable than the others in some contexts. As usual, message whitening and key blinding appear necessary if the same key is to be re-used a number of times.

17.7 Itoh's Overlapping Windows

The algorithm of Itoh et al. [4] is a sliding window technique which, in its general form, essentially includes all the preceding algorithms except that digits are non-negative. It allows any representation given by the variable base representation algorithm (Figure 17.4) subject to the base being a 2-power. As in the (second) Ha–Moon algorithm, the authors describe the conversion from binary as a recoding from left to right, enabling a table-driven exponentiation to consume the digits in the order they are generated.

The method is illustrated by several examples, the first being the overlapping windows method (O-WM) in Figure 17.11. There are two main parameters, k and h, with $k>h$ and a recommended relationship $h \geq k/2$. The base for both input and output is fixed at $m=2^{k-h}$ for this example. In the figure, the k-bit variable $Left$ consists of two parts. Its lowest $k-h$ bits are the next set of bits of D to be processed, namely the base m digit b_{i-1}. Its top h bits, the value of Top, is the remainder left from processing the more significant bits of D. The output digit d_{i-1} is no larger than $Left$ and so the digit range is from 0 to 2^k-1. Consequently, the process can be viewed as a k-bit sliding windows method with an overlap of h bits.

Figure 17.11 is just a fixed base version of the variable base recoding in Figure 17.7 with appropriate simplifications and details about "choose". It yields a left-to-right exponentiation method whose time efficiency is similar to that of m-ary exponentiation and whose space efficiency is that of 2^k-ary exponentiation. For smart card applications, k needs to be kept very small, which limits the amount of randomness which can be introduced. We need $h \geq k/2$ to add as much randomness as is in the key D.

The full O-WM method still keeps k fixed but allows h to vary, so that the base m also varies. This uses the recoding method of Figure 17.7 where Carry is chosen to keep output digits in the range 0 to 2^h-1. Interested readers should consult the original paper of Itoh [4].

17.7.1 Attacking the Algorithm

The O-WM method is very similar to the second Ha–Moon algorithm for a fixed base $m = 2^{k-h}$. The main difference is in the range for Carry in Figure 17.10. The similarity means that the same attacks are likely to work for both algorithms, although there are more complications here. The presence of zero digits and/or variable bases means that matters are easier when squares can be recognized. Then the multiplications can be correctly aligned in the same way as in Table 17.1 for Liardet–Smart. Leakage of Hamming weight enables this to be done, and so that is assumed in the leakage model here.

In particular, the attack described in Section 17.6.1 works here, using Park and Lee's averaging technique [14]. Usually a good first approximation to b_{i-1} is obtained by ignoring the effect of $Left$ on the range of randoms assigned to Top: when the previous value of Top is non-zero, $Left$ has a value of at least $m = 2^{k-h}$, which is at least 2^h if $h \le k/2$. There is a large cost in selecting $h > k/2$, but even if this were to occur, very few previous values of Top are small enough to reduce the range of the following value of Top. Hence Top has an average value only slightly less than $\frac{1}{2}(2^h-1)$, making the average value of d_{i-1} a little less than $b_{i-1} + \frac{1}{2}(m-1)(2^h-1)$. This enables an approximate value for b_{i-1} to be deduced once squaring operations in the traces have been aligned. Of course, at least 1 in 2^h of the random values will be 0, so the average for Top should be reduced by $O(2^{-h})$ and that for d_{i-1} reduced by $O((m-1)2^{-h}) = O(2^{k-2h})$. So in most cases this rough calculation should yield b_{i-1} simply by rounding down. With a bit more effort the accuracy of the digit prediction would be improved.

In comparison with earlier algorithms, it is clear that this one is more difficult to break if suitable parameters can be chosen (such as large k and small $k-h$), especially if the base is made variable. So security can be improved, but it is at the cost of run-time efficiency.

Input: h, k with $0 < h < k$, $n > 0$, $D = (b_{n-1} \dots b_2 b_1 b_0)_m$ where $m = 2^{k-h}$
Output: Random base m representation of $D = (d_{n-1} \dots d_2 d_1 d_0)_m$

$m \leftarrow 2^{k-h}$
Top $\leftarrow 0$
$i \leftarrow n$
While $i > 0$ do
Begin
    Left $\leftarrow m *$ Top $+ b_{i-1}$
    If $i = 1$ then Top $\leftarrow 0$
    else Top $\leftarrow$ Random from $\{0, 1, \dots, \min\{\text{Left}, 2^h - 1\}\}$
    $d_{i-1} \leftarrow$ Left $-$ Top
    $i \leftarrow i - 1$
End

Fig. 17.11 O-WM recoding.

17.8 Randomized Table Method

Itoh et al. [4] enhance O-WM with a "randomized table" technique (RT-WM) which modifies the digit range $\{d_{min},\dots,d_{max}\}$ to $\{r+d_{min}2^c,\dots,r+d_{max}2^c\}$ where r is a random c-bit number fixed for each exponentiation. The required pre-computed table then contains the powers of the input text under the new digit range. As the method can be applied as an additional counter-measure to any recoding scheme, the translation is described as a separate process here. However, it is a fully integrated part of the recoding in [4].

For the desired sequence of bases $m_{n-1},m_{n-2},\dots,m_0$, let

$$D_0 = r(((\dots(m_{n-2}+1)m_{n-3}+\cdots+1)m_1+1)m_0+1)$$

Compute $D' = (D-D_0-\delta)/2^c$ where $D-D_0=\delta \bmod 2^c$, and apply the chosen recoding method with the chosen bases to D' to obtain

$$D' = ((\dots(d'_{n-1}m_{n-2}+d'_{n-2})m_{n-3}+\cdots+d'_2)m_1+d'_1)m_0+d'_0$$

where the digits d'_i are in the range $\{d_{min},\dots,d_{max}\}$. Then

$$D = ((\dots(d_{n-1}m_{n-2}+d_{n-2})m_{n-3}+\cdots+d_2)m_1+d_1)m_0+d_0+\delta$$

for digits $d_i = r+d'_i2^c$ in the required range $\{r+d_{min}2^c,\dots,r+d_{max}2^c\}$.

17.8.1 Attacking the Algorithm

The motivation behind RT-WM is clearly the disruption of the digit averaging attacks described in Sections 17.6.1 and 17.7.1. Currently there are no published attacks on the method.

If the leakage were strong enough, an attack which yields any information from individual traces might be applied to the table construction phase first in order to reveal r and then applied to the exponentiation phase. This is unlikely to work without averaging over many traces: devices which employ the algorithm are likely to use hardware counter-measures which are sufficient to defeat attacks on a single trace.

The average of r is $\frac{1}{2}(2^c-1)$, which leads to an average value for D_0. In the case of Ha–Moon 2 or O-WM this would lead to average values for each digit d'_i. Ostensibly, this leads to recovery of the average for D' and hence to D. However, the borrows in computing $D' = (D-D_0-\delta)/2^c$ mean that every d'_i has the same average, namely that of a random digit. Hence the average d'_i contains no information, and D cannot be recovered in this way.

There are new methods being published which enable weak leakage to be combined successfully in the presence of randomizing counter-measures, e.g., [23]. If the same key is used sufficiently many times, the information theoretic content of the

```
m_i ← 0
If Rand(8) < 7 then
        If D ≡ 0 mod 2 then m_i ← 2 else
        If D ≡ 0 mod 5 then m_i ← 5 else
        If D ≡ 0 mod 3 then m_i ← 3
If m_i = 0 then
Begin
        p ← Rand(8)
        If p < 6 then m_i ← 2 else
        If p < 7 then m_i ← 5 else
        m_i ← 3
End
```

Fig. 17.12 One choice for digit recoding in MIST.

side channels is enough to determine the key uniquely. The only question is whether or not the information can be combined into a computationally feasible attack.

17.9 The MIST Algorithm

In the preceding algorithms, the change-of-base and variable base representation algorithms in Figures. 17.3 and 17.4 only made use of bases which are powers of 2. This means that these algorithms can be expressed as left-to-right recodings of binary, and digits can be consumed as they are generated by the usual left-to-right exponentiation algorithm. The MIST algorithm [17] deliberately selects bases which are not all powers of the same prime, but this forces digit generation to be from right to left. However, by separating digit generation from exponentiation, the exponentiation can be performed in either direction.

The original description suggests choosing the recoding base m_i randomly from the set $S = \{2,3,5\}$. An example algorithm for this is given in Figure 17.12 where $Rand(n)$ returns a random non-negative integer less than n. Because raising to the power 3 or 5 is less efficient than raising to a power of 2, the choice is biased towards base 2. However, a multiplication is saved in the exponentiation when the digit is zero, so there also a bias towards selecting bases for which the digit is 0. The digit choice mod$'$ could be the least non-negative value, but alternatives are possible, such as the residue of least absolute value. D would be stored efficiently in base 240 if the machine word were 8-bits long, so that recoding digit and base selection could be done by looking only at the lowest digit. A typical recoding example is $235_{10} = (((((0\times2+1)\times3+0)\times2+1)\times5+4)\times2+0)\times3+1$. This can be abbreviated to $235 = 1_2 0_3 1_2 4_5 0_2 1_3$ using the obvious notation to indicate the base of each digit.

Space efficiency is similar to that of binary exponentiation except for an extra register required to store one more intermediate product and space for the recoding.

Time efficiency is between that of binary and quaternary exponentiation. The details to check this require modelling the recoding as a Markov process and computing its eigenvectors [17]. The left-to-right exponentiation method of Figure 17.5 uses table entries for the multiplications. However, to achieve the best efficiency in the right-to-left method of Figure 17.6, the computations of M^{d_i} and M^{m_i} must be combined to minimize the total number of long integer multiplications. Specifically, M^{d_i} should be computed *en route* to M^{m_i}. The details can be expressed using an addition chain in which $a+b=c$ stands for the computation of $M^a \times M^b$ to obtain M^c. For example, the addition chain $1+1=2$, $2+1=3$, $2+3=5$ enables M^2, M^3 and M^5 to be computed with three multiplications, and so is suitable for base 5 with any digit except 4.

17.9.1 Attacking the Algorithm

The algorithm is designed to make it much more difficult to apply any of the previous attack methods to deduce the exponent D. Specifically, the variable base choice means there is no alignment between operations and bits of D which could be exploited. So attacks similar to those against the Liardet–Smart and Oswald–Aigner algorithms are not possible. In general, the patterns of squarings and multiplications do not seem to narrow the search space sufficiently to allow key recovery [18].

In any exponentiation, detection of operand re-use may be possible by observing Hamming weights on the bus or repeated access of the same memory locations. This makes every table-based left-to-right exponentiation potentially vulnerable. Indeed, all the previous algorithms are fatally compromised unless there is enough noise to ensure a number of mistakes in determining the operand sharing. However, in the original right-to-left MIST, the operand sharing pattern still leaves an ambiguity between the digit/base pairs 1_2 and 0_3. These occur sufficiently frequently to make it computationally infeasible to traverse the search space for the correct key D.

Nevertheless, Oswald [12] has reported analysing patterns of squarings and multiplications from a single trace by using Viterbi's algorithm [15] to select the most likely sequence of digits. This chooses about 83% of digits correctly, but apparently does not identify which are the correct ones. It is an improvement on the 74% predicted by independently selecting the most likely digits when the pattern for each digit has been identified [18]. The latter choice ignores a strong dependence between consecutive digit choices resulting from the efficiency-driven bias in Figure 17.12. With short elliptic curve keys, Oswald's technique leaves some 30 bits to modify, which is infeasible without good information about their positions. Further work on the attack may reveal this so that a prioritized search can be performed. But the result also assumes perfect information from the trace. In practice, noise leads to some degradation in the deduced pattern, and this is likely to render the attack infeasible.

17.10 Conclusions

The first attacks on exponentiation by Kocher et al. [7] showed that key recovery is possible from weak side channel information when keys are re-used with the same unprotected pattern of long integer operations. Similar trace averaging methods can reveal repeated patterns of operand re-use and of data movements in algorithms. This can still create problems for algorithms that appear to have key-independent computation patterns at the highest level [5]. Randomization techniques are required to prevent this. Key blinding provides one solution, but it may be insufficient [22]. Further randomization to confuse the attacker can be provided through the randomary exponentiation in this chapter. Most of the methods that have been developed thus far have been seen to be weak on their own and require key blinding as well if the key is to be re-used. However, for once-off key use the randomization provides the algorithms with considerably increased security and an efficiency which is often better than that of algorithms with uniformly balanced code – such as square-and-always-multiply.

17.11 Exercises

These exercises are aimed at developing an appreciation of just how difficult it is to discover the correct key from imperfect side channel leakage even when the degree of error is very low.

1. In the Liardet–Smart algorithm choose a key size of 160 bits, an upper bound $R = 4$ on the window size, and assume that there is side channel leakage from a point multiplication which provides the sequence of adds and doubles without error.
   a. What is the average number of windows which occur?
   b. Calculate the average number of different keys for which the same pattern of adds and doubles will occur.
   c. Does the algorithm become more or less secure if the value of R is altered?
   d. Is there a most secure value for R with this level of leakage?
   e. Is it computationally feasible to attack an implementation of this algorithm with so much leakage?
   f. What are the answers to these questions if the key size is doubled to 320 bits?
   g. Repeat the previous parts under the assumption that adds and doubles are only determined correctly with a probability of $p = 0.95$. (So about 10 errors may need to be corrected before guessing the digits corresponding to each addition.) Make any reasonable simplifications you wish. For example, ignore the fact that some patterns of adds and doubles are impossible.
2. This exercise involves some programming, but it can be done entirely by hand. If so, take a smaller key size such as 20 bits, adjust the probabilities,

e.g., $p_S = 1-\frac{1}{20} = 0.95$, and use the parity of word lengths in this question as the random number generator.

a. Use a random number generator to obtain a random 160-bit key D. For convenience, pick a key for which exactly 80 bits are 0 and 80 bits are 1 (including the first). Convert D into the string of squarings (S) and multiplications (M) which occur when it is used as the exponent in the usual square-and-multiply algorithm. Make sure your implementation omits unnecessary operations, such as an initial squaring of 1.
b. Suppose this key and algorithm are used in a cryptographic token which suffers from side channel leakage. Assume that $p_S = 1-\frac{1}{80}$ and $p_M = 1-\frac{1}{40}$ are the probabilities that each S and M is determined correctly before taking account of the fact that two multiplications cannot be adjacent. Use these probabilities to generate a string λ over the alphabet $\{$S,M$\}$ which might have been deduced from the trace information.
c. Write down a formula for the expected number of errors in λ. In how many cases is it expected to be possible to correct the error when "MM" occurs? Are there any end conditions which λ must also satisfy before it can correspond to the true sequence of operations? Compare your figures with those from the key and string which you generated.
d. What are the probabilities of having (a) no errors, (b) exactly one error, (c) exactly two errors, (d) exactly five errors to correct? (You may assume, for simplicity, that observable errors, such as "MM", have not occurred.)
e. Take your string λ and correct the obvious errors such as occurrences of "MM". Mark all characters in λ which might represent isolated errors. (So assume there are no adjacent errors.) How many keys need to be tested for correctness if λ contains exactly (a) one error, (b) two errors, or (c) five errors?
f. Are there any substrings which must be correct if they contain at most one error?
g. Suppose the search for the correct key is done using a machine with a 32-bit processor. Using information in earlier chapters about the cost of elliptic curve operations, estimate how many 32-bit operations are needed to check the correctness of the key if it has exactly five errors and has been used in an elliptic curve point multiplication over a 160-bit field? (Assume the test requires a 160-bit point multiplication.)

17.12 Projects

1. In this project we assume the same key is re-used without blinding, so that there are a number of traces corresponding to the same exponent. We try to reconstruct the key from these traces.

a. Write a program to generate the sequences of squarings (S) and multiplications (M) given by (a) the Liardet–Smart and (b) the Oswald–Aigner algorithms. Collect 50 such sequences, initially for 20-bit keys.
b. As in Table 17.1 align the squarings of a number of such sequences and determine whether or not the ratio of squares to multiplications is the same for each column in the averaged sequence. If it is not, can anything be deduced about the corresponding bit in the exponent?
c. Look at the patterns of S and M which are possible for an adjacent pair of bits. Do the possibilities determine either or both of the bits? Mark the bits which are determined. How many bits are doubly determined as a result of pairing them with the bits on either side? How many bits remain undetermined? If those bits had different values, would they have been determined? If so, does this mean all bits can be determined?
d. Repeat the previous part of the question, this time looking at the patterns of S and M for three adjacent bits.
e. Mechanize the bit determination process worked out in the previous parts. Apply the process to a number of sets of 2^n sequences, $n = 1, 2, \ldots$ and a 160-bit key. Calculate the average number of incorrectly determined bits for each size of trace set. If there are cases where all the bits are recovered correctly, how often is the correct key recovered for each set size?

2. a. Choose a random 160-bit key. Write a program implementing the Ha–Moon recoding algorithm of Figure 17.10 and use it to generate a number of randomized digit sequences for the key. Extend the program so that it will average the digit values at each base 4 position in the key. Use this information to predict the base 4 digit of the key as described in Section 17.6.1.
b. Apply the digit prediction process to a number of sets of 2^n sequences, $n = 1, 2, \ldots$. Calculate the average number of incorrectly determined digits for each size of trace set. If there are cases where all the digits are recovered correctly, how often is the correct key recovered for each set size?
c. Extend the digit prediction process so that it returns a probability with each digit, namely the 1 minus half the distance of $\bar{b}_i$ from its nearest integer, where $\bar{b}_i$ is the real number average used to predict the digit b_i. Repeat the previous part, this time treating each incorrect digit prediction as correct if it is among the 10% with the lowest probabilities and the next nearest integer is the correct value.
d. Estimate the cost of checking a key prediction in terms of the number of 32-bit operations required to perform a 160-bit elliptic curve point multiplication. For a case where digit errors were predictable in the sense of the previous part of the question, is it computationally feasible to correct all the digit errors identified as correctable? Decide your answer by estimating the number of hours a PC of your choice would require to test half the key possibilities. Is it computationally feasible to recover a key in this way, given that you must now divide the cost of correcting digit errors by the probability of having a key prediction that is correctable?

References

1. E. Brier and M. Joye. *Weierstraß Elliptic Curves and Side-Channel Attacks*, Public Key Cryptography (Proc. PKC 2002), D. Naccache and P. Paillier (editors), LNCS 2274, pp. 335–345, Springer-Verlag, 2002.
2. Federal Information Processing Standard 186-2. *Digital Signature Standard (DSS)*, National Institute of Standards and Technology, Maryland, USA, 2001.
3. J. C. Ha and S. J. Moon. *Randomized Signed-Scalar Multiplication of ECC to Resist Power Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 551–563, Springer-Verlag, 2002.
4. K. Itoh, J. Yajima, M. Takenaka, and N. Torii. *DPA Countermeasures by Improving the Window Method*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 303–317, Springer-Verlag, 2002.
5. M. Joye and S.-M. Yen. *The Montgomery Powering Ladder*, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 291–302, Springer-Verlag, 2002.
6. D. E. Knuth. *The Art of Computer Programming*, vol. 2, "Semi-numerical Algorithms", 2nd Edition, pp. 441–466, Addison-Wesley, 1981.
7. P. Kocher, J. Jaffe, and B. Jun. *Differential Power Analysis*, Advances in Cryptology – CRYPTO '99, M. Wiener (editor), Lecture Notes in Computer Science, 1666, pp. 388–397, Springer-Verlag, 1999.
8. P.-Y. Liardet and N. P. Smart. *Preventing SPA/DPA in ECC Systems Using the Jacobi Form*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. K. Koç, D. Naccache, and C. Paar (editors), Lecture Notes in Computer Science 2162, pp. 391–401, Springer-Verlag, 2001.
9. T. S. Messerges, E. A. Dabbish, and R. H. Sloan. *Power Analysis Attacks of Modular Exponentiation in Smartcards*, Cryptographic Hardware and Embedded Systems (Proc CHES 99), Ç. K. Koç and C. Paar (editors), Lecture Notes in Computer Science, 1717, pp. 144–157, Springer-Verlag, 1999.
10. K. Okeya and K. Sakurai. *On Insecurity of the Side Channel Attack Countermeasure Using Addition-Subtraction Chains under Distinguishability between Addition and Doubling*, Information Security and Privacy - ACISP '02, Lecture Notes in Computer Science, 2384, pp. 420–435, Springer-Verlag, 2002.
11. E. Oswald and M. Aigner. *Randomized Addition-Subtraction Chains as a Countermeasure against Power Attacks*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. K. Koç, D. Naccache, and C. Paar (editors), Lecture Notes in Computer Science, 2162, pp. 39–50, Springer-Verlag, 2001.
12. E. Oswald. *Markov Model Side-Channel Analysis* SCA-Lab Technical Report IAIK - TR 2004/03/01, Institute for Applied Information Processing and Communication, 2004. http://www.iaik.tu-graz.ac.at/research/sca-lab/index.php

13. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems, Comm. ACM 21:120–126, 1978.
14. D. J. Park and P. J. Lee. *DPA Attack on the Improved Ha-Moon Algorithm* Information Security Applications, WISA 2005, J. Song, T. Kwon, M. Yung (editors), Lecture Notes in Computer Science, 3786, pp. 283–291, Springer-Verlag, 2006.
15. A. J. Viterbi. *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*, IEEE Trans. Information Theory, **13**(2): 260–269, 1967.
16. C. D. Walter. *Sliding Windows succumbs to Big Mac Attack*, Cryptographic Hardware and Embedded Systems – CHES 2001, Ç. K. Koç, D. Naccache, and C. Paar (editors), Lecture Notes in Computer Science, 2162, pp. 286–299, Springer-Verlag, 2001.
17. C. D. Walter. *MIST: An Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis*, Topics in Cryptology – CT-RSA 2002, B. Preneel (editor), Lecture Notes in Computer Science, 2271, pp. 53–66, Springer-Verlag, 2002.
18. C. D. Walter. *Some Security Aspects of the MIST Randomized Exponentiation Algorithm*, Cryptographic Hardware and Embedded Systems – CHES 2002, B. Kaliski, Ç. K. Koç, and C. Paar (editors), Lecture Notes in Computer Science, 2523, pp. 276–290, Springer-Verlag, 2002.
19. C. D. Walter. *Breaking the Liardet-Smart Randomized Exponentiation Algorithm*, Proc. Cardis 2002, Usenix Assoc, Berkeley, CA, pp. 59–68 2002.
20. C. D. Walter. *Issues of Security with the Oswald-Aigner Exponentiation Algorithm*, Topics in Cryptology – CT-RSA 2004, T. Okamoto (editor), Lecture Notes in Computer Science, 2964, pp. 208–221, Springer-Verlag, 2004.
21. C. D. Walter and D. Samyde. *Data Dependent Power Use in Multipliers*, 17th IEEE Symposium on Computer Arithmetic – ARITH-17, IEEE Computer Society, pp. 4–12, 2005.
22. C. D. Walter. *Longer Randomly Blinded RSA Keys may be Weaker than Shorter Ones*, Information Security Applications, 8th International Workshop – WISA 2007, S. Kim, M. Yung and H.-W. Lee (editors), Lecture Notes in Computer Science, 4867, pp. 303–316, Springer-Verlag, 2008.
23. C. D. Walter. *Recovering Secret Keys from Weak Side Channel Traces of Differing Lengths*, Cryptographic Hardware and Embedded Systems — CHES 2008, E. Oswald and P. Rohatgi (editors), Lecture Notes in Computer Science, 5154, pp. 214–227, Springer-Verlag, 2008.
24. S.-M. Yen, S.-J. Kim, S.-G. Lim, and S.-J. Moon. *A countermeasure against one physical cryptanalysis may benefit another attack*, Information Security and Cryptology - ICISC 2001, K. Kim (editor), Lecture Notes in Computer Science, 2288, pp. 414-427, Springer-Verlag, 2002.
25. S.-M. Yen, C.-N. Chen, S. J. Moon, and J. C. Ha. *Improvement on Ha-Moon Randomized Exponentiation Algorithm*, Information Security and Cryptology – ICICS 2004, C. Park and S. Chee (editors), Lecture Notes in Computer Science, 3506, pp. 154–167, Springer-Verlag, 2005.

Chapter 18
Microarchitectural Attacks and Countermeasures

Onur Acıiçmez and Çetin Kaya Koç

18.1 Introduction

Microarchitectural analysis (MA) is a fast evolving area of side-channel cryptanalysis. This new area focuses on the effects of common processor components and their functionalities on the security of software cryptosystems. The main characteristic of microarchitectural attacks, which sets them aside from classical side-channel attacks, is the simple fact that they exploit the microarchitectural behavior of modern computer systems.

The fascinating progress of microprocessor technology in the last decades is maybe the most influential power that has been driving the scientific and technological advances. However, due to strictly throughput, performance, and "performance per watt"-oriented goals of modern processor designs and also "time-to-market"-driven business philosophy, the resulting products, i.e., commodity processor architectures in the market, lack a thorough security analysis. The main element that gave birth to microarchitectural analysis area is indeed this particular gap between the current processor architectures and the ideal secure computing environment.

The identification of requirements for secure execution environments has always been a challenging task since the invention of high-complexity computing devices. The security requirements of early computer systems were defined with monolithic mainframe computers in mind (cf. [9, 15, 48] and also [19] for a nice collection of early computer security efforts). Today, the domination of multi-user PC and server platforms and also the multitasking operating systems mandates a serious revision of these early requirements. Recently, we have seen an increased effort on the security analysis of daily life computer platforms. The advances in the field, more specifically, the desire to develop secure execution technologies such as AMD's Pacifica, Intel's virtualization technology (VT) and trusted execution technology (TXT) (codenamed LaGrande technology or LT for short) play an important role to increase

Samsung Information Systems America, e-mail: onur.aciicmez@gmail.com · City University of Istanbul & University of California Santa Barbara, e-mail: koc@cryptocode.net

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_18,

© Springer Science+Business Media, LLC 2009

attention on analysis of computer platform security due to [42]. Here, it has been especially shown that microarchitectural properties of modern processors create a significant security risk (cf. [3, 4, 6, 10, 29, 35]).

Today's high-end computer architectures employ several different components each of which is responsible for a specific task mostly to increase the performance of the system. Among all these different components, we will focus on only four of them in this chapter:

1. Data cache
2. Branch prediction unit
3. Instruction cache
4. Functional units, especially multiplier

These four components are the ones that had been exploited in MA until the time this chapter was written. Although it is necessary to understand the detailed functionality and purposes of these components in order to grasp the basic idea underlying the theory of MA, we cannot cover all these details in this chapter. It would take yet another book to explain even the basics of modern computer architecture, and therefore we have to assume that the reader already has at least some familiarity with computer architecture concepts. There are several books in the literature (e.g., [16, 34, 38, 39]) that give comprehensive overviews of modern computer architectures. Even though we will try to give very brief explanations of the aforementioned microprocessor components, we recommend the readers to study the related materials from [34, 39] or a similar resource in advance.

In this chapter, we cover all of these four MA types mentioned above. We start with an overview and history of microarchitectural analysis. Then we present each MA type including the basics of these attacks and examples of concrete attack strategies found in the literature. We also discuss the differences between these MA types and possible countermeasure techniques.

18.2 Overview and Brief History

The actual origins of microarchitectural analysis go back to [20, 40]. Although these publications implicitly pointed out the security risks of microprocessor components like cache, concrete and widely applicable security attacks relying on microprocessor functionalities have not been worked out until very recent years. The results of these recent studies immediately attracted significant public interest due to their implications and broad application ranges of these security breaches.

The typical targets of side-channel analysis have been and still are smart cards. However, we have seen significant increase in the research efforts spent on side-channel analysis of commodity PC platforms. Soon, the researchers realized the fact that the internal functionalities of some microprocessor components like data and instruction cache and branch prediction units cause very serious side-channel leakage and hence create crucial security risks. These efforts led to the development of microarchitectural analysis area.

Side-channel analysis can be defined as the study of the relations between the strength of cryptosystems and data-dependent variations in the so-called side-channel information, e.g., execution time and power consumption, generated during the execution of their physical implementations. Malicious parties can exploit such variations to find out the secrets used in security applications and cryptosystems. These variations either directly give the key value out during a single cipher execution or leak sensitive information which can be gathered during many executions and analyzed to compromise the system. MA attacks exploit the microarchitectural components of a processor to reveal cryptographic keys. The internal functionalities of the aforementioned processor components generate such data-dependent variations in execution time and power consumption, which are the subjects of MA.

The first type of MA we had seen is called "Cache Analysis". A cache-based attack, abbreviated to "cache attack" or "cache analysis" from here on, exploits the cache behavior of a cryptosystem by obtaining the execution time and/or power consumption variations generated via cache hits and misses, cf. [5, 6, 10–12, 24, 26, 27, 29, 31, 35, 43–45]. The potential cache vulnerability of computer systems has been known for a long time, cf. [20, 22, 23]; however, actual realistic and practical cache attacks were not developed until recent years. Cache analysis techniques enable an unprivileged process to attack another process, e.g., a cipher process, running in parallel on the same processor as done in [26, 29, 35]. Furthermore, some of the cache attacks can even be carried out remotely, i.e., over a network [6].

The current cache attacks in the literature, excluding instruction cache attacks which are fundamentally different than data cache attacks, are data-path attacks. They exploit the data access patterns of a cipher. The memory accesses of S-box-based ciphers like DES and AES are key dependent. Cache attacks analyze the cache statistics, e.g., miss/hit rates, of the cipher execution and try to reveal these memory access patterns. Cache statistics of an execution include the number of cache hits and misses, the cache lines modified by the cipher, and such. An unprivileged malicious party cannot directly obtain the cache statistics of a cipher,[1] but it can observe the side-channel leakage through execution time and/or power consumption to estimate these values. For instance, the execution time of AES software implementations is directly related to the total number of cache hits and misses occuring during an encryption, cf. [41], and someone can measure AES encryption time to determine these statistics.

Branch prediction analysis (BPA) is the second type of MA which was developed in 2006. Several variants of BPA attacks were introduced in [4], all of which exploit the side-channel leakage due to branch prediction units of microprocessors. The most powerful variant of BPA is called simple branch prediction analysis (SBPA) and it relies on the ability to run a spy process parallel to the cipher process under

[1] In fact, current processors have special registers, called performance counters, inside the chip to count and store such statistics. These registers are mainly used for performance monitoring purposes and fortunately require special privileges to be read. The potential power of a malicious party would be significantly higher without this requirement of high privilege. For further information on performance counters and performance monitoring events, refer to [49].

attack [3]. According to [3], a carefully written spy process running simultaneously with an RSA process is able to collect during one *single* RSA signing execution almost all of the secret key bits. The concept of SBPA is proved in [3] by applying an attack on the exponentiation phase of a simple RSA implementation as a case study. A spy process, which relies on the simultaneous multithreading (SMT) capability of some microprocessors, is implemented to observe the execution of an RSA cipher process. This concept was also verified by André Seznec, a well-known expert on branch prediction [50].

The actual power of SBPA is not limited to this basic application on RSA exponentiation. The SBPA has a potential to reveal the entire execution flow of a target process on *almost any* execution environment, i.e., with or without SMT. This is a very strong claim which has not been experimentally verified.

Following this interesting research field, two other MA are also introduced: exploiting instruction cache (I-cache analysis) and shared functional units (SFU analysis). Similar to BPA, I-cache and SFU analysis rely on spy routines and they reveal the execution flow of cryptosystems. In I-cache analysis, an adversary runs a spy process simultaneously or quasi-parallel with the cipher and detects the changes occuring in the instruction cache.

The principles of SFU analysis are different than the previous MA types. The previous types, i.e., cache, branch prediction, and instruction cache analysis, try to observe the changes in the *persistent state* of the mentioned microprocessor components. The spy process-oriented MA attacks, except SFU analysis, rely on the fact that the execution of cryptosystems leaves *persistent* changes in the state of shared resources like cache and branch target buffer. In other words, the cipher execution leaves "footprints" on the observable state, i.e., the so-called metadata of these resources and an unprivileged spy process can keep track of these footprints if it runs on the same processor in parallel with the cipher. An adversary can reveal the execution flow and/or the memory access patterns of cryptosystems by spying on these states and especially by detecting the changes of these states as a function of time. On the other hand, SFU analysis does not take advantage of persistent states. It follows a quite different approach and tries to detect when a certain functional unit is occupied by the cipher.

We will explain the details of each of these MA types in the following sections.

18.3 Cache Analysis

18.3.1 Basics of Cache

We can only give a brief explanation of cache in this section. We recommend the readers to explore more on cache architectures in order to grasp the details of cache attacks. For further information on cache, please refer to [17, 18, 30].

A high-frequency processor needs to retrieve the data at a very high speed in order to utilize its functional resources. The latency of a main memory is not

short enough to match this demand of high-speed data delivery. The gap between the latency of main memories and the actual demand of processors has been and will be continuously increasing as Moore's law holds. Common to all processors, the attempt to close this gap is the employment of a special buffer called cache.

A cache is a small and fast storage area used by a CPU to reduce the average memory access time. It acts as a buffer between the main memory and the processor core and provides the processor fast and easy access to the most frequently used data (including instructions) without frequent external bus accesses.

Cache stores the copies of the most frequently used data. When the processor needs to read a location in main memory, it first checks to see if the data are already in the cache. If the data are already in the cache (called a cache hit), the processor immediately uses this data instead of accessing the main memory, which has a significantly longer latency than a cache. Otherwise (a cache miss), the data are read from the memory and a copy of it is stored in the cache. This copy is expected to be used in the near future due to the temporal locality property.

A cache is partitioned into a number of non-overlapping fixed size blocks, called cache blocks or cache lines. The minimum amount of data that can be read from the main memory into a cache at once is called cache line or cache block size, i.e., each cache miss causes a cache block to be retrieved from a higher level memory. The reason why a block of data is transferred from the main memory to the cache instead of transferring only the data that are currently needed lies in spatial locality property. Since a cache is limited in size, storing new data in a cache mandates eviction of some of the previously stored data.

The method of deciding where to store and search for a data in a cache is called cache mapping strategy. Three main cache mapping strategies are direct, fully associative, and set associative mapping.

A particular data block can only be stored in a single certain location in a direct mapped cache. The exact location is determined using the address of the data block. On the contrary, a data block can be placed in potentially any location in a fully associative cache. The location of a particular placement is determined by the replacement policy. Set associative mapping is a combination of these two mapping strategies. Set associative caches are divided into a number of same size sets, called cache sets, and each set contains the same fixed number of cache lines. A data block can be stored only in a certain cache set based on the address of the data block (just like in a direct mapped cache); however, it can be placed in any location inside this set (like in a fully associative cache). Again, the particular location of a data inside its cache set is determined by the replacement policy.

The replacement policy is the method of deciding which data block to evict from the cache in order to place the new one in. The ultimate goal is to choose the data that are most unlikely to be used in the near future. There are several cache replacement policies proposed in the literature (cf. [18, 34]). In this document, we focus on a specific one: least recently used (LRU). It is the most commonly used policy and it picks the data that are least recently used among all of the candidate data blocks that can be evicted from the cache.

18.3.2 Overview of Cache Attacks

Cryptosystems have data-dependent memory access patterns. For example, S-box-based block ciphers like DES and AES employ table lookups and the indices of these lookups are functions of the plaintext and the secret key. Cache architectures leak information about the cache hit/miss statistics of ciphers through side-channels, e.g., execution time and power consumption. Therefore, it is possible to exploit cache behavior of a cipher to obtain information about its memory access patterns, i.e., indices of S-box and table lookups.

Cache attacks rely on cache hits and misses occur during the encryption/decryption process of a cryptosystem. Even if a cipher implementation has a fixed execution flow, i.e., if the same instructions are executed for any particular (plaintext, cipherkey) pair, the cache behavior during the execution causes variations in the program execution time. Cache attacks exploit such variations and narrow the exhaustive search space of secret keys.

Theoretical cache attacks were first described by Page in [31]. He characterized two types of cache attacks: trace driven and time driven. We have recently seen another type of cache attacks that can be named as "access-driven" attacks.

In trace-driven cache attacks, the adversary obtains the traces of cache activity for a sample of encryptions. We define a trace as a sequence of cache hits and misses. For example,

$$MHHMHMHM, MMHMHHMH, MMMMHHHH$$

are examples of a trace of length 8. Here H and M represent a cache hit and miss, respectively. The first memory access in the first example results in a miss, second one in a hit, and so on. If an adversary captures such traces, he can determine whether a particular access during an encryption is a hit or miss. Therefore, the adversary has the ability to observe (e.g.) if the second access to a lookup table yields a hit and can infer information about the lookup indices, which are key dependent. This ability gives an adversary the opportunity to make inferences about the secret key.

Time-driven attacks, on the other hand, are less restrictive and they do not rely on the ability of capturing the outcomes of individual memory accesses. Adversary observes the aggregate profile, i.e., total number of cache hits and misses or at least a value that can be used to approximate these numbers. For example, he measures the total execution time of a cipher and uses this measurement to approximate the number of cache misses occurring during the encryption. Note that each cache miss introduces a delay to the overall execution time and thus the total encryption time is proportional to the number of cache misses. Time-driven attacks are based on statistical inferences and therefore require much higher number of samples than trace-driven attacks.

While trace-driven and time-driven attacks analyze the outcomes of memory accesses, access-driven attacks follow a different approach. The adversary determines the cache sets that the cipher process modifies. Therefore, he can understand which

elements of the lookup tables or S-boxes are accessed by the cipher. Then, the candidate keys that cause an access to unaccessed parts of the tables can be eliminated.

In the following sections, we explain each of these cache attack types in more detail. We describe simplified attack models for each type and try to enrich the understanding of the reader by showing concrete attack examples from the literature along with these models. We can only focus on a small fraction of the previous studies on this subject in this chapter to keep the length in a reasonable range. Therefore, we first want to give a short survey on cache analysis and briefly cover the entire prior art before delving into the details of our set of concrete attack examples.

18.3.3 A Brief Survey on Cache Analysis

Although [20, 22, 23] pointed the cache vulnerability of computer systems a long time ago, actual realistic cache attacks had not been developed until recent years. D. Page described and simulated a theoretical cache attack on DES [31] in 2002. Actual cache-based timing attacks were first implemented by Tsunoo et al. [43, 44]. They developed several cache attacks on various ciphers, including MISTY1, DES, and Triple-DES [43]. Their original attack on MISTY1 was improved later in [45].

Bernstein showed the vulnerability of AES software implementations on various platforms [10]. There was a common belief that Bernstein's attack could be used as a real remote attack; however, later studies proved it wrong [27].

Osvik et al. described various local cache attack variants first in [28] in 2005, then they presented their results at CT-RSA in early 2006 [29]. They made use of a local process, called spy process, to monitor the cache activities of an AES process. They exploited the collisions between the table lookups in the first two rounds of AES and the memory access operations of the spy process. Neve et al. improved these attacks in [29] by taking the last AES round into consideration [26].

The same idea of exploiting collisions between two different processes was also used by Percival in [35] and Bertoni et al. in [11]. Percival exploited simultaneous multithreading feature of the modern processors and developed a cache attack on RSA [35]. Bertoni et al. developed a cache-based power attack on AES using the idea of exploiting external collisions.

Similar to external collisions between different processes, one can also exploit internal collisions inside a cipher. The attacks of Tsunoo et al. are based on this principle [43, 44]. Trace-driven attacks also rely on internal collisions [5, 24]. A summary of possible cache collision attacks on AES is given in [12].

None of these efforts was successful to achieve the goal of developing a generic and universally applicable cache attack that can also compromise remote systems. A remote cache attack and ideas to develop a universal remote cache attack on AES are given in [6].

Several hardware- and software-based countermeasures were proposed to prevent cache attacks in [10, 13, 29, 32, 33].

18.3.4 Time-Driven and Trace-Driven Attacks

Let P_i and K_i be the ith byte of the plaintext and cipherkey, respectively. In this chapter, each byte is considered to be either an 8-digit radix-2 number, i.e., $\in \{0,1\}^8$, that can be added in GF(2^8) using a bitwise exclusive OR operation or an integer in [0, 255] that can be used as an index.

Assume that we have a cryptosystem, which operates on a lookup table, and each element of this table is as large as the size of a cache line. There are two accesses to the same lookup table as in Figure 18.1. The indices of these accesses are a function of different bytes of the plaintext and the cipherkey. An adversary removes all of the cipher data from the cache by (e.g.) reading or writing a large piece of data. Therefore, prior to an encryption the cache does not contain any data that belongs to this cryptosystem.

Then the adversary triggers encryption with arbitrary plaintext. Whenever the two indices become equal for a plaintext, the second access will find the target data in the cache and result in a cache hit. In other words, whenever the equation

$$(P_1 \oplus K_1 = P_2 \oplus K_2)$$

holds for a plaintext, or in a different interpretation, the equation

$$(P_1 \oplus P_2 = K_1 \oplus K_2)$$

holds, then we will have a cache hit in the second during the encryption of this plaintext. Note that we assume a clean cache prior to the encryption, i.e., the cache does not contain any data belonging to this cipher. Similarly, whenever there is a cache hit in the second table lookup, these equations will also hold. Therefore, the key byte difference $K_1 \oplus K_2$ can be derived from the values of plaintext bytes P_1 and P_2 if this plaintext causes such a cache hit.

In trace-driven attacks, we assume that the adversary can directly understand if the latter access results a hit, thus he can directly obtain $K_1 \oplus K_2$.

This goal is more complicated in time-driven attacks. These attacks rely on the following facts:

- The execution time of a cipher is directly related to the number of cache misses occurring during the execution. In other words, the overall execution time of an encryption can be used to approximate the number of cache misses that occur during the encryption.

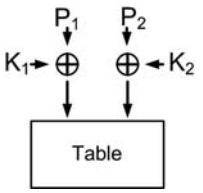

Fig. 18.1 Two different accesses to the same table.

- The average number of cache misses in encryptions of a sample of plaintext that results in a biased cache miss is different than the case of a random sample. For example, a large sample of plaintext, each of which results in a cache miss during the second table lookup, causes a different number of cache misses in average during the execution of a cipher compared to a random sample that does not result in such a biased cache miss.

The expected number of cache misses during an encryption with a plaintext that obeys the equation $(P_1 \oplus P_2 = K_1 \oplus K_2)$ is less than the expected number of cache misses when the plaintext does not obey this equation. We need to use a large sample to realize an accurate statistics of the execution. In our case, if we collect a sample of different plaintext with the corresponding execution time, the plaintext byte difference, $P_1 \oplus P_2$, that causes least number of cache misses in average (i.e., the shortest average execution time) gives the correct key byte difference.

When we consider this fact, a simple attack method becomes the following:

Phase 1: Obtain a sample of (plaintext, encryption time) pairs generated under the same target key.

Phase 2: Split this initial set into 128 subsets based on the plaintext values. In order to do this, first create a subset for each possible value of $P_1 \oplus P_2$. Note that there are 128 possibilities because P_1 and P_2 are bytes and therefore the length of $P_1 \oplus P_2$ is 8 bits. Then scan each plaintext in the initial sample and put it in the subset that corresponds to $P_1 \oplus P_2$ value of this plaintext.

Phase 3: Calculate the average encryption time of the entities in each subset. If the initial sample obtained in Phase 1 is large enough, all of these average values, except one subset, will be close to each other. The only exception is the subset that corresponds to the $P_1 \oplus P_2$ value that is equal to $K_1 \oplus K_2$. Therefore, the key byte difference can be recovered.

In this basic example attack, we assume that each element of the lookup table is as large as the size of a cache line. However, the elements of the lookup tables in real implementations are usually smaller than the cache line size. Therefore, each cache line stores more than one element of the table. Any cache miss results in the transfer of an entire cache line, not only a single element, from the main memory. This indicates the fact that there will be a cache hit even when $P_1 \oplus K_1$ is not equal to $P_2 \oplus K_2$. If a cache line stores more than one element of a table, those will be consecutive elements of the table because of the current cache architectures. Hence, we can still recover the most significant bits of $K_1 \oplus K_2$ in a real attack by following our basic attack model. This concept will be more clear when we cover our first concrete cache attack in the next section.

18.3.5 Exploiting Internal Collisions in Time-Driven Attacks

We cover some example cache attacks on AES-128 in this section. Our first example is very similar to the basic attack given above.

The most widely used AES software implementation employs five different lookup tables. There are 10 rounds of computations in AES-128 and 16 table lookups in each round. The indices of the first round table lookups are in the form $P_i \oplus K_i$ for $i \in \{0, 1, ..., 15\}$. The indices to the first AES table in the first round are $P_0 \oplus K_0$, $P_4 \oplus K_4$, $P_8 \oplus K_8$, and $P_{12} \oplus K_{12}$.

We can directly apply the idea given in the previous section to these indices. If we use the simple attack method on the first two indices $P_0 \oplus K_0$ and $P_4 \oplus K_4$ by splitting our initial sample set into the subsets based on the value $P_0 \oplus P_4$, the distinct average encryption time values will indicate the value of $K_0 \oplus K_4$.

Figure 18.2 shows the average execution time for each subset that was formed during the search of $K_0 \oplus K_4$ in a real attack using the indices $P_0 \oplus K_0$ and $P_4 \oplus K_4$. It can clearly be seen in this figure that there are several distinct points as opposed to a single point for the correct $K_0 \oplus K_4$ value. The reason, as explained above, is that the elements of the lookup tables in real implementations are usually smaller than the actual cache line size and thus each cache line stores more than one element of

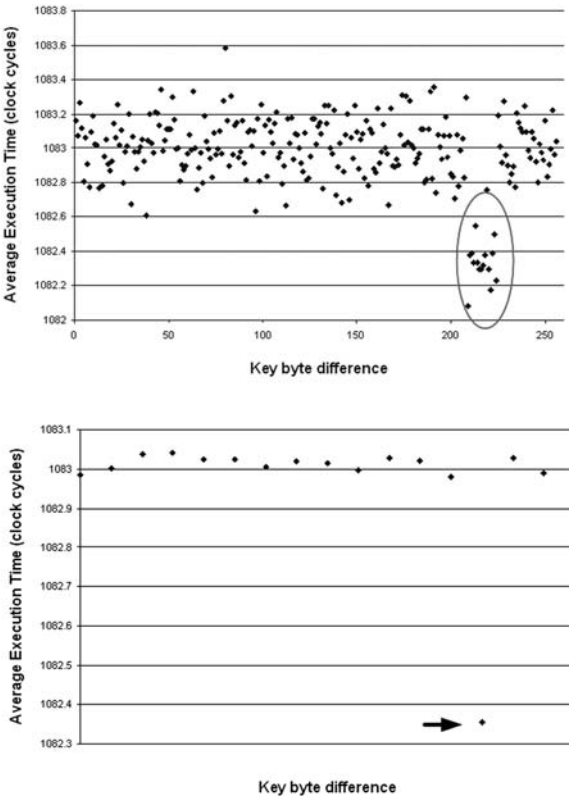

Fig. 18.2 The first graph shows the results of the search for $\langle K_0 \oplus K_8 \rangle_{msp}$ using the indices $(P_0 \oplus K_0)$ and $(P_8 \oplus K_8)$ in an experimental attack. The second graph shows the same search when the number of table elements in a single cache line is known to be 16.

the table. Therefore, there is a cache hit during the latter table lookup whenever the most significant parts of $P_0 \oplus K_0$ and $P_4 \oplus K_4$ become equal for a plaintext. So, we can only find the difference of the most significant part of the key bytes using the equation

$$\langle P_0 \rangle \oplus \langle P_4 \rangle = \langle K_0 \rangle \oplus \langle K_4 \rangle$$

where $\langle A \rangle$ stands for the most significant part of A. The size of the most significant parts, i.e., the number of bits that can be recovered by exploiting these two indices, depends on how many table elements a cache line holds.

The dashed ellipse in the top graph of this figure obviously contains more than 8 and less than 32 points, so an attacker can conclude that there are 16 table elements in a cache line, which also means that the most significant parts of the key byte differences are 4 bits long. Using less sets during a search on key byte differences gives more clear identification of the correct value, because each set contains more elements in this case. An increase in the size of a sample gives a better estimation of the expected execution time for this sample because the variance of the average encryption time decreases proportional to the size of the sample. The bottom graph shows the average execution time of each 16 sets that can be formed for the same search.

Applying the same idea on different indices of the first round, we can find the key byte differences $\langle K_i \oplus K_{4*j+i} \rangle$ with $i, j \in \{0,1,2,3\}$. This attack is called first round attack. On current widely used processors the search space can typically be reduced to 56, 68, or 80 bits for 128-bit keys. If we also consider the indices of the second round table lookups, called second round attack or two-round attack, we can reduce the search space to 32 bits. The equations used in second round attack are more complicated than first round attack; therefore, we do not cover them in detail in this chapter. Further details can be found in [6].

These attack principles can be used to develop a remote cache attack, i.e., a cache attack that does not require local access to the target machine and can compromise the systems over a network just by sending messages to the systems and measuring the response time. They also showed how one can devise and apply such remote cache attacks on other cryptosystems. Their experiments indicate that cache attacks can be used to extract secret keys of remote systems if the system under attack runs on a server with a multitasking or multithreading system and a large enough workload. Although a large number of measurements are required to successfully perform a remote cache attack, it is feasible in principle.

18.3.6 Access-Driven Attacks

In a computer system, we have a main memory, which stores the data of each process running on the system, and a cache between the memory and the CPU as shown in Figure 18.3. We represent each cache block with a square in this figure and each column corresponds to a different cache set. For example, the cache in this figure has two blocks in each column, so it is two-way set associative. Assume that the data

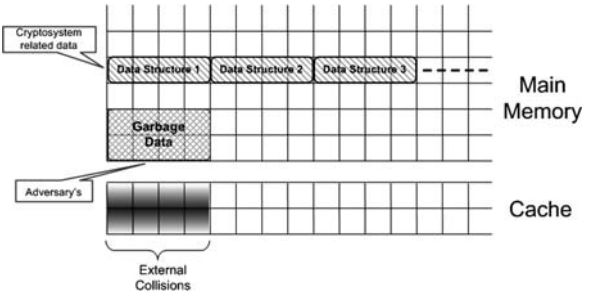

Fig. 18.3 Main memory and cache structure in a CPU.

blocks in a column of the memory map only to the corresponding cache set in the same column of the cache. Mapping a memory block to a cache set means that this particular cache block can only be stored in that set of the cache. As an example, the garbage data and data structure 1 can only be in the dark area of the cache in Figure 18.3. In fact, the mapping between memory locations and cache blocks in real computer systems are different and more complicated than our basic assumption. However, for the sake of simplicity and clarity, we follow this assumption in this chapter.

Also assume the following. There are two different processes, the cryptosystem and a malicious process, called Spy process, running on the same machine. Cryptosystem process operates on several different data structures. The actual value of the secret key affects which of these data structures (e.g., which parts of a table lookup) are accessed during an encryption and when (e.g., in which round) they are accessed.

An adversary can easily understand if the cipher has at least one access to, for example, data structure 1 during a particular encryption. This is due to the situation that accesses to garbage data and data structure 1 create external collisions. We define a collision in this context as the situation that occurs when an attempt is made to store two or more different data items at a location that can hold only one of them. We use the term "external collision" if these data items belong to different processes. On the other hand, if the data items belong to the same process, we call it as "internal collision".

In our case here, the cache does not have enough number of sets to store the garbage data and data structure 1 at the same time. Since the cache is only two-way associative and the garbage data completely fills the dark area in, an access to garbage data results in replacing any previously stored data in the dark area. Similarly, an access to any data that maps to the same cache location also replaces some or all of the garbage data if it resides in the cache prior to this access. Therefore, an access to the garbage data may evict data structure 1 from the cache and vice versa. This fact enables an adversary to devise an attack on the cryptosystem process.

A basic attack works as the following. An adversary reads the garbage data which would force the CPU to load the content of it into the cache (Figure 18.4a). Then

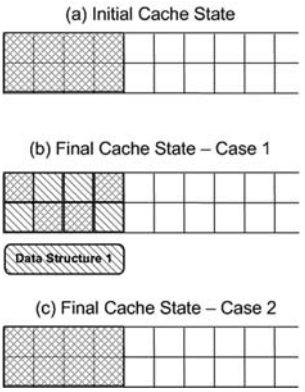

Fig. 18.4 Steps of a basic attack.

the adversary triggers an encryption and the cryptosystem is run under this initial cache state. There are two possible cases that may happen during the encryption:

Case 1: Cipher accesses data structure 1.
Case 2: Cipher does not access data structure 1.

When the first case happens, the access to data structure 1 changes the content of the first four cache sets as shown in the figure. Otherwise, these sets remain unchanged. In Case 1 (Case 2, resp.) the final state of the first four cache sets just after the encryption becomes like Figure 18.4b (Figure 18.4c, resp.). When the adversary reads the garbage data again after the encryption, he can understand which case was true, because reading the garbage data creates some cache misses and thus takes longer in Case 1. Similarly, at least in theory, the adversary can use the same technique for other data structures and reveal the entire set of items that are accessed during an encryption. Since this set depends on the secret key value, he can gain invaluable information to narrow the exhaustive search space.

This model describes an active attack where the adversary must be able to control the contents of the cache. The cache attacks that rely on this basic model correspond to access-driven types. Percival's attack on RSA [35] (cf. Section 18.3.7), Osvik et al. and Neve et al.'s attacks on AES [26, 28, 29] (cf. Section 18.3.6.1), and the power attack by Bertoni et al. [11] use this attack model. In fact, branch prediction analysis (cf. Section 18.4) and instruction cache analysis (cf. Section 18.5) are also based on similar approaches.

18.3.6.1 Osvik–Shamir–Tromer (OST) Attacks

Our first example of access-driven cache attack was developed by Osvik, Shamir, and Tromer. They described and simulated several different methods on AES to perform local cache attacks. We cover only their most powerful attack in this chapter. The principle of their attack is very similar to the above basic model. They apply

the same idea on AES to determine which parts of AES tables are accessed during an AES operation.

In their attack, the adversary runs a spy process, which reads a local array to load it into the cache. After the spy process sets the state of the cache to a known state by forcing the CPU to replace the cache entries with the data of this array, the adversary triggers an AES encryption with a known plaintext. Immediately after the execution of AES, the spy process takes over the CPU and starts reading the same array again. However, this time it also measures the time it takes to read the blocks of this array. That way, as explained above, the adversary can determine which parts of the array got evicted from the cache. The parts of this local array that are not evicted by AES process directly give out which blocks of AES tables were not accessed during the encryption.

Let us consider the index $P_0 \oplus K_0$ as an example. This attack is a known plaintext attack and we know the values of P_0 but trying to find K_0. When we apply this attack, we determine the elements of AES tables that are not accessed during the encryption of this plaintext, i.e., the values that $P_0 \oplus K_0$ cannot be equal to. We can then eliminate the values of K_0 that would cause an access to these unaccessed elements for this particular value of P_0. If we can gather enough data from several encryptions, we end up with the correct value of $\langle K_0 \rangle$. Remember that we cannot reveal the least significant part of K_0 for the reason explained in the previous section. In general, if we consider the first round indices, we can reveal $\langle K_i \rangle$ for $0 \leq i \leq 15$. If we extend our focus on the second round indices, then we can recover the entire key.

Osvik et al. described several variations of access-driven cache attacks on AES. They also suggested relying on hardware-assisted SMT feature to detect the changes of the cache states on-the-fly during the encryption. However, their attack idea does not require SMT feature and can principally work on any multitasking environment.

Figure 18.5 shows real experimental results of an access-driven variant taken from [29]. In this experiment, Osvik et al. observed the cache activity of several encryptions with random but known plaintext values. They ran the spy process and collected "measurement scores" for each possible value of $\langle K_i \rangle$ as the following. They collected samples of the form (P, y, m) consisting of arbitrary table indices y, random plaintexts P, and measurement scores m. The measurement scores are the time delays when the spy reads the blocks of its local array that map to the

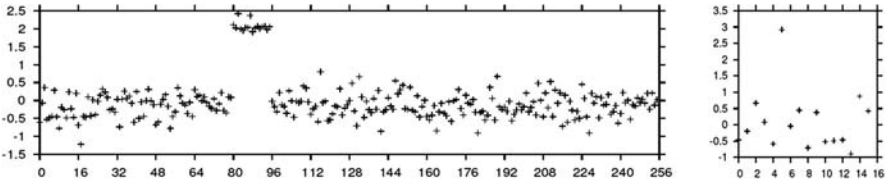

Fig. 18.5 Average measurement scores of first round OST attack for 30,000 (left) and 800 (right) triggered encryptions under the same key. The x-axis shows $P_5 \oplus y$ (left) and $\langle P_5 \oplus y \rangle$ and the y-axis shows the average measurement scores in units of clock cycles. The actual value of $\langle K_5 \rangle$ can easily be determined as 0×5.

same cache set as the block of the AES table that contains table element with the index y. If we consider a key byte $\langle K_i \rangle$, whenever $\langle y \rangle$ becomes equal to $\langle P_i \oplus K_i \rangle$ for a particular P, the measurement score m will be higher for this (P, y) pair. If we collect measurement scores (P, y, m) for a sample of known plaintext, split these scores into different subsets based on $\langle P_i \oplus y \rangle$ values and calculate the average m values in each of these subsets, then the subsets that correspond to the correct $\langle K_i \rangle$ value will have higher average measurement scores compared to the other subsets. This is exactly what Osvik et al. did to generate the results shown in Figure 18.5.

18.3.7 Percival's Hyper-Threading Attack on RSA

Percival developed a cache attack on RSA, which relies on hardware-assisted SMT capability [35]. Our attack model described in Section 18.3.6 can work on almost any system. But, such access-driven attacks become much more powerful on simultaneous multithreading environments, because the adversary can run the spy process simultaneously with the cipher. Running these processes simultaneously allows an attacker to obtain not only the set of data structures accessed by the cipher but also the approximate time that each access occurs.

In Percival's attack, the adversary again runs a spy process but this time it is run simultaneously with the server on an SMT processor. Spy process has a local array just like the previous attack and continuously reads each block of this local array in the same order. Note that each of these blocks correspond to a different cache line. The spy process reads the blocks that map to the same cache set together and measures the overall read time for each of these sets. In other words, the spy process observes each cache set (via reading the local array in a structured manner) in a certain order to determine whether the RSA process modifies this cache set. If reading a cache set takes longer, the attacker can conclude that this set was accessed during the time interval between the last read of the set by the spy process and the current read.

The experimental results of Percival's attack are given in Figure 18.6. The color of each little square in this figure indicates the time it takes the spy to read the corresponding cache set, denoted as cache congruency class. These colors can be considered as the measurement scores for each cache set. All of the squares in a particular column map to the same cache set. The vertical axis shows the time of the observation.

In general, such an attack can reveal the "footprints" of a victim process. Percival applied this idea on RSA and was able to identify the order of squaring and multiplication operations in OpenSSL's RSA implementation. Percival's attack on OpenSSL's sliding windows exponentiation (with a window size of 5 bits) could reveal an average 200 bits of information about each of the two 512-bit secret exponents.

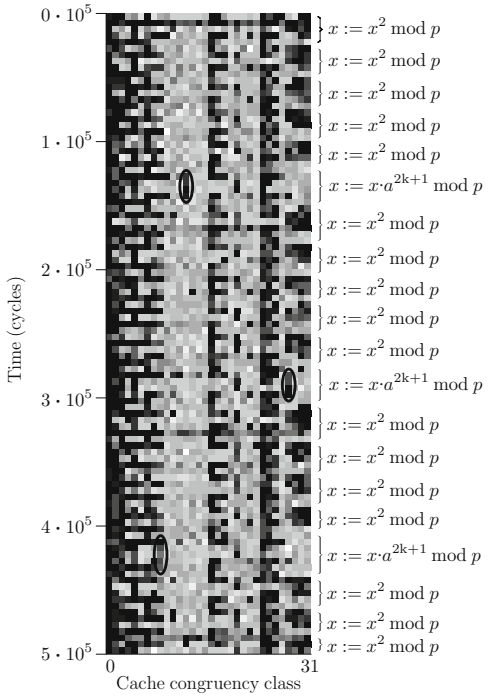

Fig. 18.6 Graphical representation of a small portion of the spy measurements in Percival's attack on RSA.

18.4 Branch Prediction Analysis

Branch prediction analysis (BPA) is the second microarchitectural analysis type we cover in this chapter. Several methods to exploit the branch prediction mechanism are developed [3, 4]; branch prediction mechanism is nowadays a part of all general purpose processors. We call all of these attacks branch prediction analysis. The most powerful variant of BPA is called simple branch prediction analysis (SBPA) and it is our subject in this section. Please refer to [4] for other variants of BPA.

18.4.1 The Concept of Branch Prediction

Deep microprocessor pipelines coupled with the ability to fetch and issue multiple instructions at every machine cycle led to the development of superscalar processors. Superscalar processors target a theoretical or best-case throughput of less than one machine cycle per completed instructions, cf. [39]. However, the actual throughputs of superscalar processors are limited by the available instruction-level parallelism (ILP) in the executed code. When branch instructions were recognized as one

of the most crucial performance killers of superscalar processors, microprocessor architects quickly invented the concept of branch predictors in order to circumvent those performance bottlenecks. There has been very significant amounts of research on more and more sophisticated branch prediction mechanisms, cf. [34, 38, 39]. However, it turns out that the branch mechanisms can be exploited to attack the integrity of the processes running on the processor.

Superscalar processors rely on branch prediction mechanisms to execute instructions speculatively to overcome control hazards, cf. [34, 39]. Therefore, the actual performance of microprocessors is greatly affected by the efficiency of speculation, which makes it one of the key issues in modern superscalar processor design.

A *branch instruction* is a point in the instruction stream of a program where the next instruction is not necessarily the next sequential one. There are two types of branch instructions: unconditional branches (e.g., jump instructions, goto statements) and conditional branches (e.g., if-then-else clauses, for and while loops). For conditional branches, the decision to take or not to take the branch depends on some condition that must be evaluated in order to determine the correct execution path. During this evaluation period, the processors speculatively execute instructions from one of the possible execution paths instead of stalling and awaiting for the decision to come through.

The key to achieve higher processor performance is the ability to predict the correct execution path as accurately as possible. The ultimate goal of branch prediction mechanisms is therefore to predict the most likely execution path in such a case. The accuracy of branch prediction mechanisms greatly affects the overall performance. Thus, it is very beneficial if the branch prediction algorithm tries to predict the most likely execution path in a branch. If the prediction is true, the execution continues without any delays. However, if the speculatively executed instruction flow turns out to be wrong, which is called a *misprediction*, the instructions on the pipeline that were speculatively issued have to be dumped and the execution has to start over from the mispredicted instruction path, thus suffers from a misprediction delay. Measurable timing differences between a correct and incorrect prediction are exactly what the BPA/SPBA attacks capitalize on.

A microprocessor needs the following information to speculatively execute the instructions after a conditional branch:

- *The outcome of the branch prediction.* It has to determine the outcome of a conditional branch, i.e., whether it needs to be taken or not taken, in order to execute the correct instruction sequence. This outcome depends on the evaluation of the condition and it is not immediately available when a conditional branch is issued. The processor must execute the branch and check the outcome of the condition, which is evaluated in later stages of the pipeline. Instead of waiting the *actual* outcome of the branch to come through, the processor starts executing a possible instruction sequence, which is predicted as the correct sequence by the branch prediction unit (BPU). This prediction is usually based on the history of the same branch as well as the history of other branches executed just before the current branch, cf. [39].

- *The target address of the branch.* If the BPU predicts a conditional branch to be taken, the instructions in the target address have to be fetched and issued. In this case, the processor needs the address of the predicted instruction sequence, i.e., the target address, in order to fetch and issue these instructions. Similar to the outcome of the branch, the target address may not be immediately available too. Therefore, the CPU tries to keep records of the target addresses of previously executed branches in a buffer, the so-called branch target buffer (BTB).

Overall common to all *branch prediction units (BPU)* is the following Figure 18.7. As shown, the BPU consists of mainly two "logical" parts, the BTB and the predictor. The predictor is that part of the BPU that makes the prediction on the outcome of the branch under question. The BTB is the buffer where the CPU stores the target addresses of the previously executed branches. Since this buffer is limited in size, the CPU can store only a number of such target addresses, and previously stored addresses may be evicted from the buffer if a new address needs to be stored instead. If the processor cannot find a target address in BTB (called a BTB miss), the execution suffers from a BTB miss delay similar to a branch misprediction. Further details of branch prediction can be found in [34, 38, 39].

18.4.2 Simple Branch Prediction Analysis

Several methods to exploit the branch prediction mechanisms are developed [4]. One of these methods presented relies on the fact that processors keep the target addresses of recently executed conditional branches in BTB. This attack, which was initially named as "trace-driven BTB attack", was significantly improved in [3] and renamed as simple branch prediction analysis (SBPA).

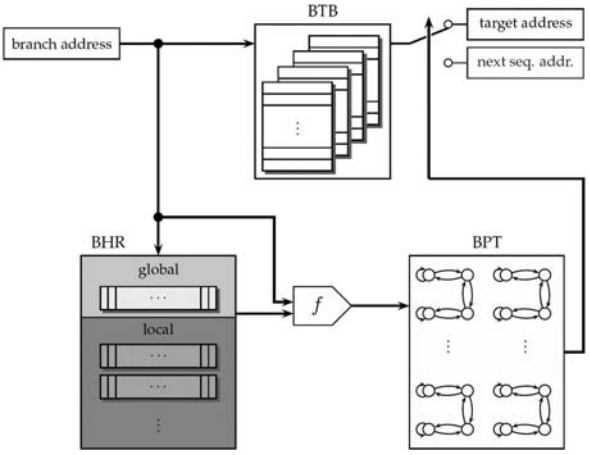

Fig. 18.7 Branch prediction unit architecture.

SBPA is also a spy-oriented attack similar to the cache eviction attacks presented above. However, there is a fundamental difference between SBPA attacks and pure data cache eviction attacks. Attacking the BTB is targeting the instruction flow, which is more complicated than the data flow within the memory hierarchy, i.e., between the L1 data cache and the main memory.

The SBPA attack is applied on RSA by running a spy process simultaneously with RSA on a multithreaded processor. It could reveal the execution flow of the RSA process by observing the BTB state transitions during a single RSA operation. Although the attack is carried out on an SMT system, it is argued that this attack can be used on almost any processor [7].

SBPA takes advantage of the fact that both processes share the BTB. The spy process continuously executes a certain fixed sequence of a sufficiently higher number of branches to guarantee the eviction of a target branch executed by the cryptographic process. In [3], the conditional branch of RSA, which determines the square and multiply sequence of the exponentiation, was targeted. When the next time the cryptographic process executes the target conditional branch, the address of this branch cannot be found in BTB. If the cipher takes this branch, the processor must rewrite the target address back to BTB, which causes the footprint of this branch to be left in BTB. Since the spy process continuously executes several branches, it will detect this change shortly after it happens. This way, the spy process can observe the traces of the target branch in terms of "taken" and "not taken".

We have to mention that the branches executed by the spy maps to the same BTB area with the target branch. In other words, the spy process intentionally creates conflicts (thus a race condition) between its branches and the target branch. Therefore, whenever the target branch turns out to be taken, the target address of this branch needs to be stored in BTB by evicting one of the spy branches from the BTB. When the spy process re-executes its branches, it will encounter a misprediction on the branch that has just been evicted. This misprediction will also trigger further mispredictions because the entry of the evicted spy branch needs to be re-stored and another not-yet-re-executed spy branch entry has to be evicted, which will also cause other mispredictions. Overall, the execution time of this spy step suffers from *many* misprediction delays resulting in a high timing gap between taken and not-taken situations of the target branch.

Reference [3] demonstrated the first SBPA attack on the S&M algorithm implemented in OpenSSL-0.9.7. They showed that measurements taken from a *single* run of the S&M exponentiation is sufficient to extract almost all of the RSA secret exponent bits. Figure 18.8 shows their experimental results. The y-axis shows the measurements of the spy in terms of the clock cycles and x-axis shows the order of these measurements, which also indicates the order of the RSA operations. As you can see, the spy measurements become clearly different when RSA executes a multiplication compared to the case of a square operation. Since the order of the spy measurements is also known, we can easily extract the operation sequence of RSA and construct the secret exponent from these measurement results.

Attacking the S&M algorithm was only a case study to show the potential of SBPA. The actual potential of SBPA is much broader as stated in [3, 7]. In general,

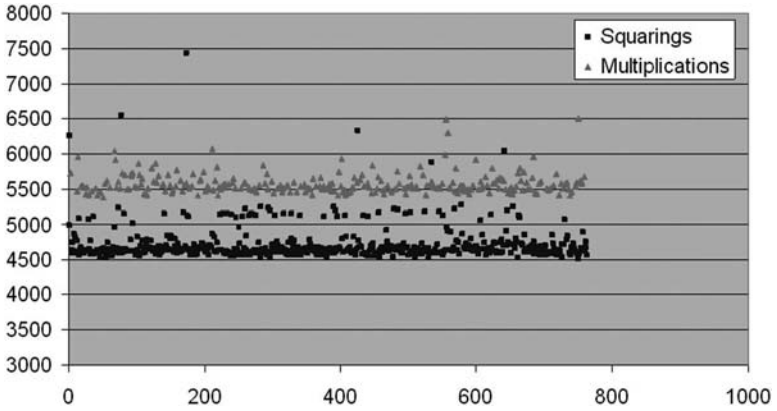

Fig. 18.8 Experimental results of SBPA on S&M exponentiation, yielding 508 out of 512 secret key bits.

SBPA can reveal the execution flow of a process and thus endangers any system if their execution flow depends on sensitive information. Several potential application areas of SBPA are given in [3]. Reference [7] identifies a novel side-channel attack on binary extended euclidean algorithm, which is enabled by the SBPA methodology.

18.5 I-cache Analysis

Instruction cache was identified as another microarchitectural analysis source [2]. This new technique, called instruction cache analysis or shortly I-cache analysis, is also spy oriented and tries to reveal the execution flow of a process just like branch prediction attacks.

Many processors use different caches for data and code segments of a process. Instruction cache (I-cache) stores recently executed instructions from the code segment and when a process starts executing a code block that is not currently stored in the cache, i.e., in case of a cache miss, the processor loads these instructions from main memory into the cache. Since a cache is limited in size, several different code blocks share the same cache sets/lines. In case of a cache conflict between different code blocks, they evict each other from the instruction cache when their executions are interleaved. As you can see, the functionality of instruction cache is very similar to data cache and also shared between different processes. Since we already explained above some details of cache architectures and the general functionality, which is very similar in both data and instruction cache, we do not cover these concepts again in this section.

In [2], the consequences of cache conflicts are exploited by creating intentional conflicts between the instructions of an RSA process and a spy code and forcing the

processor to evict the RSA instructions out of I-cache. The attack scenario given in [2] is the following.

A spy process runs simultaneously with the cipher process and tries to determine which instructions are executed by the cipher. It spies on the cipher execution by observing the I-cache state transitions. Assume that the spy process tries to understand when the cipher "touches" a certain I-cache set during the execution of a part of cipher code. The spy process continuously executes a set of "dummy" instructions that map to this particular cache set. These dummy instructions are not intended to perform any useful calculations other than filling some I-cache space, i.e., the "spied-on" cache set. These dummy instructions fill completely and precisely this I-cache set, no more, no less. Therefore, the processor has to store them into the spied-on cache set, which inevitably causes the eviction of the previous entries in that I-cache location and sets it to a known predetermined state. When the cipher executes some instructions that map to this particular I-cache location, the predetermined state set by the spy process must change. The spy process can determine this change via the timing difference when it re-executes its dummy instructions.

Reference [2] applied this basic principle to OpenSSL's RSA implementation. Due to some performance improvement reasons, OpenSSL first calls either multiplication or square functions from its multiprecision library during a Montgomery operation and then calls Montgomery reduction function to reduce the result to the modulus. This technique causes key-dependent sequence of multiplication and square function calls during sliding window exponentiation, which is the default exponentiation algorithm used in OpenSSL. In the experiments of [2], the spy continuously executed dummy instructions to evict the instructions of the multiplication function and measured the execution time of its own dummy instructions as described above. Note that, in this practical attack, the spy does not observe a single I-cache set but a number of sets that can hold the instructions of the multiplication function.

Figure 18.9 shows the results of this experiment. Again the y-axis shows the measurements of the spy in terms of the clock cycles and x-axis shows the order of

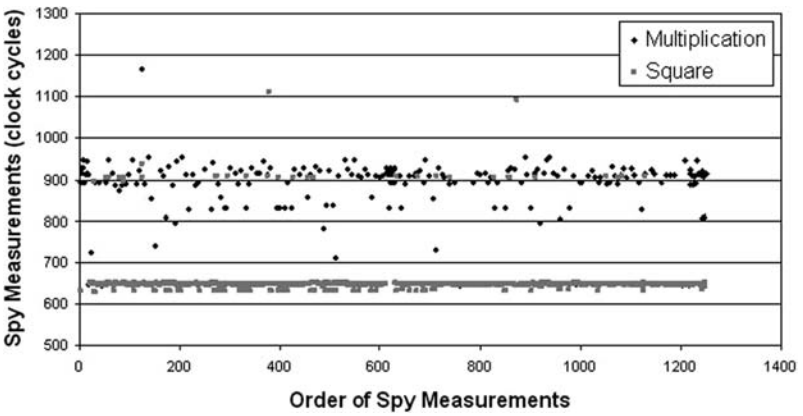

Fig. 18.9 Experimental results of I-cache analysis given in [2].

these measurements, which also indicates the order of the RSA operations. These timing measurements were taken during *a single RSA operation* under a random 1024-bit key. As you can see in this figure, the operation sequence of RSA could be successfully observed via I-cache analysis.

18.6 Exploiting Shared Functional Units

All of the microarchitectural analysis types we have covered above rely on the fact that there are some buffers (e.g., data cache, instruction cache, BTB) shared between different processes running on the same processor. The state of these buffers and the transitions between these states are affected by the execution of any of these processes and also affect the execution of other processes. These attacks can be applied on any platform with or without SMT capability as explained in [26]. However, [1] presents a novel attack which is very unique to certain SMT architectures and it seems that it cannot be carried out on CPU architectures without SMT hardware assistance. In this sense, this attack is unique because it does not rest upon a shared resource with the persistent state property between context/process switches. Instead, it is based upon the fact that some SMT technologies share complex functional units between the hardware-supported threads, i.e., between logical processors within a physical SMT processor, in order to keep the SMT area over head cheap.

Reference [1] presents an attack on OpenSSL which exploits the fact that the integer multiplier in Pentium 4 architecture is shared between the two threads executing on the same SMT-enabled processor. Since a multiplier does not preserve any persistent state, this attack methodology is quite different than other microarchitectural types. The principle idea of this attack is the following. A spy process continuously executes a number of dummy multiplication instructions and measures their execution time. Whenever the other process (RSA process in this case) performs some multiplications by executing its own multiplication instructions, the time measured by the spy will be longer. This is because the spy and the cipher instructions race to occupy the shared multiplier. When the multiplier executes a multiplication instruction from the cipher process, the spy multiplication instructions have to wait their turn until the multiplier finishes its current task, which eventually cause longer execution time for the spy instructions and can easily be detected by the spy.

Reference [1] also exploited the key-dependent RSA implementation of OpenSSL. As we already explained above, OpenSSL implementation has key-dependent sequence of multiplication and square function calls during sliding window exponentiation. We need to mention another important aspect of OpenSSL's implementation. RSA operations make use of multiprecision multiplication routines due to their long operand sizes. Usually the operands in RSA exponentiations are 512 bits or 1024 bits long, respectively, for 1024 and 2048 bit RSA keys. Note that RSA implementations usually benefits from Chinese remainder theorem and operates on half-sized operands compared to the size of entire public keys. Multiprecision libraries represent large integers as a sequence of machine-sized words.

OpenSSL implements two different multiplication algorithms: Karatsuba and "normal" multiplication. OpenSSL uses Karatsuba multiplication to multiply two numbers with an equal number of words (e.g., square operation). Karatsuba multiplication takes $O(n^{\log_2 3})$ time, where n is the number of words in the operands, cf. [25]. When multiplying two numbers with an unequal number of words of size n and m, OpenSSL executes normal multiplication, which runs in $O(nm)$ time. Therefore, a square operation takes less time than a multiplication in OpenSSL. This particular way of implementing RSA causes the leakage of operation sequence because the execution time variations depend on this sequence due to the difference between the implementations of multiplication and square operations.

Shared functional unit attack uses this timing difference, which can be observed by the spy process as described above, to distinguish between modular multiplications and squares. The operation sequence reveals the entire secret key in a binary square and multiply exponentiation. In case of OpenSSL's sliding window exponentiation, a large number of key bits can be derived.

18.7 Comparing Microarchitectural Analysis Types

Data cache attacks try to reveal the data access patterns of cryptosystems. On the other hand, branch prediction, I-cache, and shared function unit attacks expose the execution flow of the ciphers. Implementations with fixed instruction flow, which is usually the case for block ciphers, are intrinsically protected against these attacks. However, public key cryptosystem implementations, e.g., those of RSA and ECC, usually have key-dependent operation flow. It is possible to implement these systems without key-dependent execution flow, but it comes with some degree of performance loss. Due to such performance and optimization reasons, the developers usually choose to implement these systems in a way that cause execution flow variations, which make BPA, I-cache, and SFU attacks a real threat to actual security systems.

It is also possible to determine the execution flow of a cipher (e.g., RSA) by analyzing the data access patterns as done in [35]. However, implementations can avoid this threat by carefully handling the layout of data structures on the memory. For example, OpenSSL changed the way it handles the RSA structures to avoid data cache attacks. Even when the data structures are handled in a special way, BPA, I-cache, and SFU analysis can compromise the implementations if the execution flow remains key dependent. Similarly, data cache attacks can be applied on implementations with fixed execution flow if the data access patterns are key dependent. Therefore, both data and instruction cache analysis must be considered during the design and implementation of security critical systems.

The basic difference between I-cache and branch prediction analysis is the following. Branch prediction analysis presented in [3, 4] specifically targets conditional branches. A conditional branch controls the execution of different instruction paths. Thus, the outcome of a conditional branch, which can be observed via

BPA, leaks the instruction path to an adversary. However, using conditional branch is only one way to implement execution flow control. There are other techniques, which may be protected against BPA, to conditionally alter the execution flow without the use of conditional branches. In this sense, I-cache analysis is broader than BPA because it reveals the execution flow regardless of how execution flow control is implemented. For example, [8] proposes to use indirect jumps instead of conditional branches as a countermeasure to BPA, but this mitigation is still vulnerable to I-cache analysis.

The main difference between shared FU attack and the other MA types is the fact that it does not exploit the footprints of cryptosystems that are left on the persistent states of a buffer. Other MA types, data cache, I-cache, and BPA attacks rely on the persistent states of some buffers shared between different processes running on the same processor. These attacks can be applied on any platform with or without SMT capability. On the other hand, shared FU attack is based upon the fact that some SMT technologies share complex functional units between the hardware-supported threads and thus it seems that it cannot be carried out on CPU architectures without SMT hardware assistance.

18.8 Countermeasures for Microarchitectural Analysis

In this section we investigate possible countermeasures to prevent MA threats. We cover mainly software-based countermeasures and very briefly mention possible hardware-based countermeasures. The reason why we do not cover hardware-based approaches in greater length is because there had not been any real attempt, unfortunately, to employ such hardware changes in real systems. Therefore, we focus on more practical aspects of MA prevention, more specifically software mitigation methods, in this section.

Several countermeasures for AES were proposed in [13] against cache attacks. Particularly, [13] argues that permuting the AES lookup tables prevents access-driven attacks. They also propose to use smaller lookup tables, e.g., original AES S-box, during the first and last rounds of AES computations. Current cache attacks on AES exploit first, second, and the last round accesses. Using smaller tables during these rounds make it more difficult, i.e., require more samples, to apply these cache attacks. A formal study was presented in [41] to analyze the effects of table sizes, among other parameters, on the performance of time-driven attacks.

The only cache attack on RSA is from Percival [35]. He exploited the fact that OpenSSL implementation accesses different data structures during square and multiplication operations. In other words, Percival's attack can extract the operation sequence of RSA by tracing the cache activities, cf., Section 18.3.7. Moreover, it is also possible to determine which table entries are used in a multiplication step because table entries were stored in consecutive regions of memory and thus they map to different cache sets. Reference [14] proposes an implementation technique that does not have these weaknesses. To be more precise, [14] proposes to change

the memory layout of RSA exponentiation table and to interleave the table entries in memory instead of storing them in a consecutive manner. This way, each access to any table entry results in touching the same cache lines, which makes the accesses to different table entries indistinguishable. However, one still can observe the operation sequence of RSA due to the simple fact stated above. Therefore, [14] also suggests to employ fixed window exponentiation, which has a constant operation sequence. These countermeasures were implemented in OpenSSL as an optional protection mechanism, i.e., a user has the option to turn these mechanisms on.

Branch prediction and I-cache attacks exploit execution flows of cryptosystems. The best mitigation method for these attacks is to implement cryptosystems with fixed execution flows. Reference [7] analyzed the strength of OpenSSL's RSA implementation considering branch prediction analysis and detected several weak points that needed to be changed. Particularly, [7] suggested to remove several conditional branches that affect the strength of RSA implementation. OpenSSL team took these suggestions into consideration and modified the implementations. Reference [7] also proposes a new method to implement high-level execution flow of RSA without any variations. A similar proposal is also given in [21].

Another mitigation method for branch prediction vulnerabilities is given in [8]. They suggest to implement conditional branches via indirect branching. In other words, their method comprises storing the addresses of the branch legs in memory and loading the corresponding address into a register during runtime based on the evaluated condition of a conditional branch and using this register as the jump target. Since they do not consider to avoid execution flow variations, the proposed mitigation does not truly provide high security. Their protection can easily be overcome by I-cache attacks.

There are also several hardware countermeasures in the literature proposed to prevent microarchitectural attacks. We do not cover the details of hardware-based countermeasures in this book. The interested readers can refer to [10, 32, 33, 47].

18.9 Exercises

1. What is (are) the difference(s) between time-driven and access-driven cache attacks? Which one is more efficient and why?
2. What is (are) the fundamental difference(s) between data cache and instruction cache attacks?
3. Both I-cache analysis and simple branch prediction analysis (SBPA) can very effectively reveal the execution flow of a cryptosystem. For example, both attacks can extract the sequence of multiplication/square operations in an RSA exponentiation by observing only a single run of the cipher. However, I-cache analysis is more general than SBPA in the sense that I-cache analysis can compromise SBPA-resistant systems. What is the reason of this situation?
4. Which cryptosystems are susceptible to branch prediction analysis?

5. Why is it easier to carry out microarchitectural attacks on simultaneous multithreading processors?
6. Which microarchitectural attack type does seem to work only on simultaneous multithreading processors? Why?

18.10 Projects

1. There are some reference cache attack codes in [10] and [35]. Verify Bernstein's attack on AES and Percival's attacks on RSA using these reference codes.
   Bernstein's AES attack mimics a remote cache attack. In real remote attacks, the timing measurements must be obtained by a client and they also contain a lot of noise due to network delays, etc. However, in Bernstein's experiments, the encryption time is measured inside the crypto process, i.e., the server, instead of in the client. Modify Bernstein's reference code and measure the actual response times inside the client and verify that this attack becomes practically infeasible in a realistic remote attack.
   Percival's attack, the experiments need to be performed on a simultaneous multithreading processor. You can run these experiments on an HT-enabled Pentium 4. In order to synchronize spy and crypto processes, you can use `pthread` library.
2. There are several publications in the literature, cf. [36, 37, 46], that give the details of how one can extract the secret exponent in an RSA exponentiation by observing the occurrences of extra reduction steps in Montgomery multiplication during the exponentiation. In other words, if we can observe which multiplication/square operations perform an extra reduction step during an entire modular exponentiation with a secret exponent, we can extract the value of this secret exponent by using some statistical analysis techniques. These techniques can also tolerate some levels of errors in the observations. SBPA and I-cache analysis are useful tools to perform such observations to detect the occurrences of extra reduction steps. Study [36, 37, 46], find an RSA implementation that employs Montgomery multiplication with extra reduction step (OpenSSL v.0.9.8e would suffice), and try to apply SBPA and I-cache attacks on this implementation to detect the extra reduction steps and extract the secret exponent. Also analyze the error rates in the observations and their effect on the necessary sample size.
3. Branch prediction attacks have been demonstrated only on conditional branches so far. Avoiding conditional branches in cryptographic implementations are thought to prevent these attacks. However, there are still many open questions. For example, it may also be possible to compromise security systems by detecting when and which unconditional branches are executed during the course of a cryptographic operation. Such information can be useful if the implementation has data-dependent execution time and/or control flow variations. Another problem is due to error detection purposes. The software implementations, whether cryptographic operations or regular applications, have many conditional branches to detect run-time errors/anomalies. For example, a widely accepted convention

is to use conditional branches, e.g., if-then-else statements, to check the return values of functions/methods to detect run-time errors, unexpected results, etc. Although these conditional branches have a low probability to alter the execution flow, they can be exploited via branch prediction analysis. These aspects need further investigations.

References

1. O. Acıiçmez and J.-P. Seifert. Cheap hardware parallelism implies cheap security. *4th Workshop on Fault Diagnosis and Tolerance in Cryptography — FDTC 2007*, pp. 80–91, IEEE Computer Society, 2007.
2. O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, pp. 11–18, ACM Press, 2007.
   Also available at: Cryptology ePrint Archive, Report 2007/164, May 2007.
3. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. On The power of simple branch prediction analysis. *2007 ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'07)*, R. Deng and P. Samarati, editors, pp. 312–320, ACM Press, 2007.
   Also available at: Cryptology ePrint Archive, Report 2006/351, October 2006.
4. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pp. 225–242, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007, also available at: Cryptology ePrint Archive, Report 2006/288, August 2006.
5. O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on AES (Short Paper). 8^{th} *International Conference on Information and Communications Security — ICICS06*, P. Ning, S. Qing, and N. Li, editors, pp. 112–121, Springer-Verlag, Lecture Notes in Computer Science series 4307, 2006. Full version is available at: Cryptology ePrint Archive, Report 2006/138, April 2006.
6. O. Acıiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, M. Abe, editor, pp. 271–286, Springer-Verlag, Lecture Notes in Computer Science series 4377, 2007.
7. O. Acıiçmez, S. Gueron, and J.-P. Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. 11^{th} *IMA International Conference on Cryptography and Coding*, S. D. Galbraith, editor, pp. 185–203, Springer-Verlag, LNCS 4887, 2007, also available at: Cryptology ePrint Archive, Report 2007/039, February 2007.
8. G. Agosta, L. Breveglieri, I. Koren, G. Pelosi, and M. Sykora. Countermeasures Against Branch Target Buffer Attacks. *4th Workshop on Fault Diagnosis and Tolerance in Cryptography — FDTC 2007*, pp. 75–79, IEEE Computer Society, 2007.

9. D. E. Bell and L. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corporation, 1973.
10. D. J. Bernstein. Cache-timing attacks on AES. Technical Report, 37 pages, April 2005. Available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
11. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES power attack based on induced cache miss and countermeasure. *International Symposium on Information Technology: Coding and Computing - ITCC 2005*, vol. 1, pp. 4–6, 2005.
12. J. Bonneau and I. Mironov. Cache-Collision Timing Attacks against AES. *Cryptographic Hardware and Embedded Systems — CHES 2006*, L. Goubin and M. Matsui, editors, pp. 201–215, Springer-Verlag, Lecture Notes in Computer Science series 4249, 2006.
13. E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, February 2006.
14. E. Brickell, G. Graunke, and J.-P. Seifert. Mitigating Software Side Channels in AES and RSA Software. Developers track RSA 2006, RSA conference San Jose, 2006.
15. Department of Defence. *Trusted Computing System Evaluation Criteria (Orange Book)*. DoD 5200.28-STD, 1985.
16. R. C. Detmer. *Introduction to 80X86 Assembly Language and Computer Architecture*. Jones & Bartlett Publishers, 2001.
17. P. Genua. A Cache Primer. Technical Report, Freescale Semiconductor Inc., 16 pages, 2004. Available at http://www.freescale.com/files/32bit/doc/app_note/AN2663.pdf.
18. J. Handy. *The Cache Memory Book*. 2nd edition, Morgan Kaufmann, 1998.
19. NIST. History of Computer Security Project: Early Papers. National Institute of Standards and Technology, Computer Security Division: Computer Security Resource Center, available at http://csrc.nist.gov/publications/history/index.html
20. W. M. Hu. Lattice scheduling and covert channels. *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 52–61, IEEE Computer Society, 1992.
21. M. Joye and M. Tunstall. Securing OpenSSL Against Microarchitectural Attacks. *International Conference on Security and Cryptography — SeCrypt'07*, J. Hernando, E. Fernndez-Medina, and M. Malek, editors, pp. 189–196, INSTICC Press, 2007.
22. J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, vol. 8, pp. 141–158, 2000.
23. P. C. Kocher. Timing Attacks on Implementations of Diffie–Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology – CRYPTO '96*, N. Koblitz, editor, pp. 104–113, Springer-Verlag, Lecture Notes in Computer Science series 1109, 1996.
24. C. Lauradoux. Collision attacks on processors with cache and countermeasures. *Western European Workshop on Research in Cryptology — WEWoRC 2005*, C. Wolf, S. Lucks, and P.-W. Yau, editors, pp. 76–85, 2005.

25. A. J. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, New York, 1997.
26. M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. *13th International Workshop on Selected Areas of Cryptography — SAC'06*, E. Biham and A. M. Youssef, editors, pp. 147–162, Springer, Lecture Notes in Computer Science series 4356, 2007.
27. M. Neve, J.-P. Seifert, and Z. Wang. A refined look at Bernstein's AES side-channel analysis. *Proceedings of ACM Symposium on Information, Computer and Communications Security — ASIACCS'06*, p. 369, ACM Press, 2006.
28. D. A. Osvik, A. Shamir, and E. Tromer. Other People's Cache: Hyper Attacks on HyperThreaded Processors. Presentation available at http://www.wisdom.weizmann.ac.il/∼tromer/.
29. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology — CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, D. Pointcheval, editor, pp. 1–20, Springer-Verlag, Lecture Notes in Computer Science series 3860, 2006
30. R. van der Pas. Memory Hierarchy in Cache-Based Systems. Technical Report, Sun Microsystems Inc., p. 28, 2002, available at http://www.sun.com/blueprints/1102/817-0742.pdf
31. D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.
32. D. Page. Defending Against Cache Based Side-Channel Attacks. Technical Report. Department of Computer Science, University of Bristol, 2003.
33. D. Page. Partitioned Cache Architecture as a Side Channel Defence Mechanism. Cryptography ePrint Archive, Report 2005/280, August 2005.
34. D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. 4th edition, Morgan Kaufmann, 2006.
35. C. Percival. Cache missing for fun and profit. *BSDCan 2005*, Ottawa, 2005. Available at http://www.daemonology.net/hyperthreading-considered-harmful/
36. W. Schindler. A Combined Timing and Power Attack. *PKC 2002*, D. Naccache and P. Paillier, editors, LNCS 2274, pp. 263–279, 2002.
37. W. Schindler and C. D. Walter. More Detail for a Combined Timing and Power Attack against Implementations of RSA. *9th IMA International Conference on Cryptography and Coding*, K. G. Paterson, editor, pp. 245–263, Springer-Verlag, LNCS Nr. 2898, 2003.
38. T. Shanley. *The Unabridged Pentium 4 : IA32 Processor Genealogy*. Addison-Wesley Professional, 2004.
39. J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill, 2005.
40. O. Sibert, P. A. Porras, and R. Lindell. The Intel 80× 86 Processor Architecture: Pitfalls for Secure Systems. *IEEE Symposium on Security and Privacy*, pp. 211–223, 1995.
41. K. Tiri, O. Acıiçmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. *14th International Workshop on Fast Software*

Encryption — FSE 2007, A. Biryukov, editor, pp. 399–413, Springer, Lecture Notes in Computer Science series 4593, 2007.
42. Trusted Computing Group. http://www.trustedcomputinggroup.org.
43. Y. Tsunoo, T.Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. *Cryptographic Hardware and Embedded Systems — CHES 2003*, C. D. Walter, Ç. K. Koç, and C. Paar, editors, pp. 62–76, Springer-Verlag, Lecture Notes in Computer Science series 2779, 2003.
44. Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. *ISITA 2002*, 2002.
45. Y. Tsunoo, E. Tsujihara, M. Shigeri, H. Kubo, and K. Minematsu. Improving cache attacks by considering cipher structure. *International Journal of Information Security*, vol. 5(3), pp. 166–176, Springer-Verlag, 2006.
46. C. D. Walter and S. Thompson. Distinguishing Exponent Digits by Observing Modular Subtractions. *Topics in Cryptology — CT-RSA 2001, The Cryptographers' Track at the RSA Conference 2001*, D. Naccache, editor, LNCS 2020, pp. 192–207, 2001.
47. Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. *34th International Symposium on Computer Architecture — ISCA'07*, pp. 494–505, ACM Press, 2007.
48. W. Ware. Security Controls for Computer Systems. Report of Defense Science Board Task Force on Computer Security; Rand Report R609-1, The RAND Corporation, 1970.
49. Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide.
50. A. Seznec. Branch Prediction under Scrutiny for Possible Security Flaw available online at
http://www.irisa.fr/activity/new/007/branchpredictionattack004.

Authors' Biographies

Çetin Kaya Koç

Çetin Kaya Koç received his Ph.D. in Electrical and Computer Engineering from University of California Santa Barbara in 1988. He was an assistant professor at University of Houston (1988–1992) and assistant, associate and full professor at Oregon State University (1992–2007). He established Information Security Laboratory at OSU and guided 14 Ph.D. students, 8 of who are currently professors. In September 2001, he received Oregon State University Research Award for outstanding and sustained research leadership.

His research interests are in algorithms and architectures for cryptography, computer arithmetic and embedded systems. He has co-founded *Workshop on Cryptographic Hardware and Embedded Systems* (chesworkshop.org) in 1999 and has been the program chair and proceedings editor from 1999 to 2003. He is now a permanent member of the steering committee of CHES. Recently, he has also co-founded a new conference, *International Workshop on the Arithmetic of Finite Fields* (waifi.org), which is a forum of engineers and mathematicians interested in efficient software and hardware realizations of finite fields. He has co-authored one book, *Cryptographic Algorithms on Reconfigurable Hardware*, published by Springer. He has been an associate editor of *IEEE Transactions on Computers* and *IEEE Transactions on Mobile Computing* and guest co-editor of two issues (April 2003 and November 2008) of *IEEE Transactions on Computers* on cryptographic and cryptanalytic hardware and embedded systems. Dr. Koç published more than 120 journal, conference and book articles, 7 US patents, and edited 5 books. He is an *IEEE Fellow* since 2007 *for contributions to cryptographic engineering*.

Dr. Koç is currently with City University of Istanbul and University of California Santa Barbara.

Onur Acıiçmez

Onur Acıiçmez holds a Ph.D. in Electrical Engineering and Computer Science from Oregon State University. During his Ph.D. research, he discovered several microarchitectural attacks, a number of security weaknesses in widely used cryptographic software, and participated in strengthening these systems against microarchitectural cryptanalysis. He received his B.S. degree in Computer Engineering and Information Science in July 2002 from Bilkent University in Ankara, Turkey, where he has received several awards and distinctions. He completed his Master's study focusing on high performance implementations of cryptographic algorithms in Electrical and Computer Engineering in May 2004 at Oregon State University. He has authored and co-authored 22 patent applications and several papers on side-channel analysis, microarchitectural cryptanalysis, and trusted computing. He is currently working on trusted computing as a research scientist at Samsung Information Systems America, an R&D center of Samsung Electronics.

Sandro Bartolini

Sandro Bartolini is an assistant professor at the Department of Information Engineering, University of Siena, Italy. He received his Ph.D. in Computer Engineering from the University of Pisa, Italy. He took part in various research and industrial projects on the following topics, which constitute his main research interests: embedded systems, cache and memory subsystems, compiler optimizations, link/post-link profile-driven tuning of applications, low-power techniques, advanced computer architecture, and special hardware for security and cryptography. He is a member of the European HiPEAC (High Performance and Embedded Architecture and Compilation) network of excellence and is currently working in the SARC FP6 integrated project on "scalable computer architecture". Dr. Bartolini is associate editor of the *EURASIP Journal on Embedded Systems* and was guest editor of *ACM Computer Architecture News* and of the *Journal of Embedded Computing*. He is coorganizer of the MEDEA International Workshop since 2002. He is a member of IEEE, IEEE Computer Society, and ACM.

Nigel Boston

Nigel Boston grew up in England and attended Cambridge and Harvard. His postdoctoral work in Paris and Berkeley was followed by 12 years at the University of Illinois, except for 6 months as Rosenbaum Fellow at the Newton Institute in Cambridge, UK, when he witnessed Wiles's announcement of a proof of Fermat's last theorem. In recent years he has moved toward engineering, becoming founding director of the Illinois Center for Cryptography and Information Protection. In 2002, he was hired by the University of Wisconsin-Madison as part of the computational sciences cluster, with joint appointments in Mathematics and Electrical and

Computer Engineering. In 2006–2007 he was Williams-Hedberg-Hedberg chair at the University of South Carolina. Beginning in September 2008 he will be Stokes professor of pure and applied algebra at University College Dublin, Ireland.

Debrup Chakraborty

Debrup Chakraborty received Bachelor of Mechanical Engineering degree from Jadavpur University, Kolkata, India, in 1997. He obtained M.Tech. and Ph.D. degrees in Computer Science from Indian Statistical Institute, Kolkata, India, in 1999 and 2005, respectively. Currently he is a postdoctoral researcher in the Computer Science Department of Centro de Investigaciones y Estudios Avanzados del IPN, Mexico City, Mexico. Dr. Chakraborty's current research interests include design and analysis of provably secure symmetric encryption schemes, efficient software/hardware implementations of cryptographic primitives, and pattern recognition.

Pawel Chodowiec

Pawel Chodowiec received the B.Sc. degree in Telecommunications from Warsaw University of Technology in Warsaw, Poland, and the M.Sc. degree in Electrical and Computer Engineering from George Mason University in Fairfax, VA, USA. During his research at George Mason University he cooperated with Dr. Kris Gaj on development of novel hardware implementations for cryptographic algorithms with emphasis on block ciphers. Currently he works at Mantaro Networks, Inc., as an FPGA engineer and consultant with specialization in high-speed network and security applications.

Matthew Darnall

Matthew Darnall is a Ph.D. student in Mathematics at the University of Wisconsin – Madison. He received his B.A. in Mathematics at Humboldt State University in Arcata, CA. His research interests are cryptography, number theory, and irregularities of distribution.

Serdar Süer Erdem

Serdar Süer Erdem received the B.S. degree from Boğaziçi University, Istanbul, Turkey, in 1992, the M.S. degree from the Pennsylvania State University in 1996, and the Ph.D. degree from the Oregon State University in 2002, all in Electrical and Computer Engineering. Currently, he is an assistant professor at the Electronics Engineering Department of Gebze Institute of Technology in Kocaeli, Turkey. His

research interests include computer arithmetic, cryptography and network security, embedded systems.

Kris Gaj

Kris Gaj received the M.Sc. and Ph.D. degrees in Electrical Engineering from Warsaw University of Technology in Warsaw, Poland. He was a founder of Enigma, a Polish company that generates practical software and hardware cryptographic applications used by major Polish banks. In 1998, he joined George Mason University, where he currently works as an associate professor, doing research and teaching courses in the area of cryptographic engineering and reconfigurable computing. His research projects center on novel hardware architectures for secret key ciphers, hash functions, public key cryptosystems, and factoring, as well as development of specialized libraries and application kernels for high-performance reconfigurable computers. He has been a member of the Program Committees of CHES, CryptArchi, and Quo Vadis Cryptology workshops, and a General Co-Chair of CHES 2008 in Washington, D.C. He is an author of a book on breaking German Enigma cipher used during World War II.

Roberto Giorgi

Roberto Giorgi is an associate professor at Department of Information Engineering, University of Siena, Italy. He was research associate at the University of Alabama in Huntsville, USA. He received his Ph.D. in Computer Engineering and his Master's in Electronics Engineering, magna cum laude both from University of Pisa, Italy. He is participating in the European projects HiPEAC (High Performance Embedded-system Architecture and Compiler) and SARC (Scalable ARChitectures) in the area of future and emerging technologies. He took part in ChARM project, developing software for performance evaluation of ARM processor-based embedded systems with cache memory, for VLSI Technologies Inc., San Jose. His current interests include computer architecture themes such as embedded systems, multiprocessors, memory system performance, workload characterization. He has been selected by the European Commission as an independent expert for evaluating the European project SHAPES (Scalable Software Hardware Architecture Platform for Embedded Systems). He is a member of ACM and a senior member of the IEEE, IEEE Computer Society.

Marc Joye

Marc Joye received his Ph.D. degree in applied sciences (cryptography) from the Université Catholique de Louvain, Belgium, in 1997. In 1998 and 1999, he was a postdoctoral fellow of the National Science Council, Republic of China. From 1999

to 2006, he was with the Card Security Group, Gemplus (now Gemalto), France. Since August 2006, he has been with the Security Laboratories, Thomson R&D, France. His research interests include cryptography, computer security, computational number theory, and smart card implementations. He is author and co-author of more than 80 scientific papers and holds several patents. He served in numerous program committees and was program chair for CT-RSA 2003, CHES 2004, and ACM-DRM 2008. He is a member of the IACR and co-founder of the UCL Crypto Group.

Enrico Martinelli

Formerly with the Italian National Research Council as a researcher at the Istituto di Elaborazione dell'Informazione, Pisa, presently Enrico Martinelli is a full professor of the University of Siena, where he teaches courses on logic design and information security and is the head of the Department of Information Engineering. His main research interests are focused on logic design of high performance digital structures for special applications as digital signal processing and cryptography.

Francisco Rodríguez-Hénriquez

Francisco Rodríguez-Hénriquez received his bachelor's degree in Electrical Engineering from the University of Puebla, Mexico, in 1988. He obtained M.Sc. degree in Electrical and Computer Engineering from the National Institute of Astrophysics, Optics and Electronics (INAOE), Mexico, in 1992 and he received Ph.D. degree in Electrical and Computer Engineering, Department of Oregon State University in 2000. Currently, he is an associate professor (CINVESTAV-3B researcher) at the Computer Science Department of CINVESTAV-IPN, in Mexico City, Mexico, which he joined in 2002. Dr. Rodríguez-Hénriquez's major research interests are in cryptography, finite field arithmetic, and hardware implementation of cryptographic algorithms.

Pankaj Rohatgi

Pankaj Rohatgi is a research staff member and the manager of the Internet Security Group at IBM's TJ Watson Research Center. He received a B.Tech degree in Computer Science and Engineering from IIT Delhi in 1988 and a Ph.D. in Computer Science from Cornell University in 1994. From 1993 to 1996 he worked at Thomson R&D Labs and at the Sun-Thomson Interactive Alliance as a security architect for the OpenTV operating system. In 1996, he joined the IBM TJ Watson

Research Center where he has contributed to products such as the IBM System S, the IBM 4758 crypto co-processor, and conducted research in the areas of applied cryptography, side-channel cryptanalysis, network and systems security, security for embedded systems, and security risk management. He has published over 35 scientific articles and holds 11 patents. He has given numerous invited talks in the area of side-channel cryptanalysis. He is currently serving as the program co-chair of the 2008 Workshop on Cryptographic Hardware and Embedded Systems (CHES).

Gökay Saldamlı

Gökay Saldamlı received his B.S. and M.Sc. degrees from the Mathematics Department of Middle East Technical University, Ankara, Turkey, in 1996 and 2000, respectively. He completed his Ph.D. degree in Electrical and Computer Engineering at Oregon State University in June 2005. He worked 2 years at Samsung Electronics, Giheung, South Korea, as a senior engineer. Currently, he is working at Eczacıbaşı Embedded Design Center, in Istanbul, Turkey, and leading various projects related to cryptography and information security. His research interests include cryptographic engineering, public-key cryptography, low-power cryptography, and computer arithmetic. He is a member of the IEEE Computer Society and the International Association of Cryptologic Research (IACR).

Erkay Savaş

Erkay Savaş received the B.S. (1990) and M.S. (1994) degrees in Electrical Engineering from the Electronics and Communications Engineering Department at Istanbul Technical University. He completed the Ph.D. degree in the Department of Electrical and Computer Engineering at Oregon State University in June 2000. He worked for various companies and research institutions before joining Sabanci University as an assistant professor in 2002. He is the director of the Cryptography and Information Security Group (CISec) of Sabanci University. His research interests include cryptography, data and communication security, privacy in biometrics, trusted computing, security and privacy in data mining applications, embedded systems security, and distributed systems. He is a member of IEEE, ACM, the IEEE Computer Society, and the International Association of Cryptologic Research (IACR).

Werner Schindler

Werner Schindler received his Diploma in Mathematics in 1989, his Ph.D. in Mathematics (Dr. rer. nat.) in 1991, and his postdoctoral lecture qualification (Habilitation) in 1998, all at Darmstadt University of Technology, Germany. Since 1993, he has been working as a federal civil servant at Bundesamt für Sicherheit in der Informationstechnik (BSI) in Bonn, Germany. He is also an adjunct professor of

mathematics (außerplanmäßiger professor) at Darmstadt University of Technology since 2005. His main fields of expertise are cryptographic algorithms and protocols, side-channel attacks on smart cards and software implementations, random number generators for cryptographic applications, electronic payment systems, stochastic simulations, measure and integration theory.

François-Xavier Standaert

François-Xavier Standaert was born in Brussels, Belgium, in 1978. He received the Electrical Engineering degree and Ph.D. degree from the Université Catholique de Louvain, respectively, in June 2001 and June 2004. In 2004–2005, he was a Fulbright visiting researcher at Columbia University, Department of Computer Science, Network Security Lab (September 04 to February 05), and at the MIT Medialab, Center for Bits and Atoms (February 05 to July 05). In March 2006, he was a founding member of IntoPIX s.a. He is now a postdoctoral researcher of the Belgian Fund for Scientific Research (FNSR) at the UCL Crypto Group. His research interest includes digital electronics and FPGAs, cryptographic hardware, design of symmetric cryptographic primitives, physical security issues, and side-channel analysis.

Berk Sunar

Berk Sunar received his B.Sc. degree in Electrical and Electronics Engineering from Middle East Technical University in 1995 and his Ph.D. degree in Electrical and Computer Engineering (ECE) from Oregon State University in December 1998. After briefly working as a member of the research faculty at Oregon State University, Sunar has joined Worcester Polytechnic Institute as an assistant professor. Since July 2006, he is serving as an associate professor. He is currently heading the Cryptography and Information Security Laboratory (CRIS). Sunar received the National Science Foundation CAREER award in 2002. He served as the co-editor of Cryptographic Hardware and Embedded Systems Workshop (CHES) 2005 and International Workshop on the Arithmetic of Finite Fields (WAIFI) 2007. His research interests include lightweight and tamper-resilient cryptography. Sunar is a member of the IEEE Computer Society, the ACM, and the International Association of Cryptologic Research (IACR) professional societies.

Colin Walter

Colin Walter is a senior member of the IEEE, a member of the IACR CHES steering committee, and head of cryptography at Comodo, one of the main certificate

authorities. He obtained his doctorate in algebraic number theory from Cambridge University and worked in academia prior to his current post. He helped design several cryptographic ASICs in the late 1980s. From this stemmed a series of papers on hardware arithmetic, particularly on Montgomery modular multiplication. Amongst these is the earliest design for a fully systolic modular multiplier with purely local connections. In the latter 1990s he was a consultant working on the Mondex purse to defeat timing and power attacks. This led to a number of papers on side-channel attacks, many of which assume standard blinding techniques and showing, paradoxically, that longer keys may be cryptographically weaker. His proposals for counter-measures include a randomized exponentiation algorithm.

Tuğrul Yanık

Tuğrul Yanık received a B.S. in Computer Engineering from the Aegean University in Izmir, Turkey, in 1996, an M.Sc. in Computer Science and Engineering from the Oregon Graduate Institute of Science and Technology in 1999, and a Ph.D. degree from the Oregon State University in 2002. He worked for 4 years as a software engineer at Mentor Graphics Corp. Currently he is an assistant professor at the Computer Engineering Department of Fatih University in Istanbul, Turkey. His research interests include computer arithmetic, cryptography, and network security focusing on VOIP security and security protocols. He is a member of IEEE.

Index

5-tuple, 12
7-tuple, 12
$B(1,p)$, 29, 38, 39, 44, 45, 47, 48, 51
$\Phi(\cdot)$, 31
χ^2 test, 44, 45, 47
χ^2-distribution, 44
q-dependent random variables, 31, 34, 35
(O1), 42, 45
(O2), 43
(O3), 43, 45
(O4), 43, 45
(R1), 6, 9, 25, 49
(R2), 8, 9, 13, 25, 49
(R3), 10, 13, 20, 25, 39, 49
(R4), 12, 13, 25
/dev/random, 19
/dev/urandom, 19

access-driven, 480
access-driven attacks, 485
add with carry, 78
AddRoundKey, 237, 285
Adversarial Model, 385
adversary, 28
AES, 236
AES key scheduling, 286
AES round, 282
Affine transformation, 239
AIS 20, 49
AIS 31, 49
algorithmic postprocessing, 26
almost modular reduction, 137
almost spectral reduction, 138
Amplitude Modulated, 410
Angle Modulated, 410
Angle Modulation, 422
Arithmetic and Logic Unit (ALU), 296
ASIC, 247, 251
Authenticated Encryption, 338
authenticated encryption mode, 3
autocorrelation, 36
autocovariance, 36
average Hamming-weight, 374

Baby-Step Giant-Step Attack, 181
Baggini and Bucci, 58
Barrett Modular Reduction, 82, 92
Barrett modular reduction, 83
base, 127
base polynomial, 127
basic test, 45
BCH codes, 75
Bernolli number, 147
Bezout's identity, 102
Big Mac Attack, 452
binary extension field, 110
Binary Extension Fields, 87
binomial, 162, 163
Bit tracing, 373
Bitstream Security, 313
Blum-Blum-Shub DRNG, 11, 20
Branch Prediction Analysis (BPA), 490
branch prediction unit, 476
Branch Prediction Unit (BPU), 492
Branch Target Buffer (BTB), 492
Bucci-Luzzi Testable TRNG, 64

cache analysis, 477, 478
cache attacks, 477
CBC-MAC, 354
CCM, 3, 359
CCM Authentication, 348, 353
CCM Encryption, 349
CCM mode, 347

Cipher Block Chaining Mode, 333
Cipher Feedback Mode, 334
circular matrix, 241
Classical Template Attacks, 389
CMC, 341
CMC, EME, EME*, 342
Coefficient Reduction, 98
Comba's method, 217
computational number theory, 509
computational security, 9, 49
conditional entropy, 17, 29, 35, 41, 51
Conditional Subtractions, 440
confidence interval, 30
Counter Mode, 335
countermeasures, 365, 376, 428
CPLD, 55
CRT, 167
cryptographic keys, 2
cryptographic postprocessing, 41, 49
cryptographically secure RNG, 11
CTR Mode, 355
CWC, 341
cycles per instruction (CPI), 225
cyclotomic polynomial, 159

das bit, 26, 39
das bits, 14
das random number, 26, 27, 47, 49
das random numbers, 14
data cache, 476
Data randomizations, 310
DeMoivre-Laplace approximation, 40
designer, 26–28, 42, 45–48
deterministic RNG, 7
Dichtl and Golić RNG, 67
DIEHARD, 57
Differential power analysis, 373
digitized analog signal, 26
Direct Emanations, 409
Discrete Fourier Transform, 129
Discrete Fourier Transform (DFT), 126
Disk Encryption, 339
DRNG, 7, 25
DSA, 41
dual-field adder, 107, 112
dual-field adder (DFA), 216
dual-field arithmetic, 2
Dual-Radix Multiplier, 116

EAX, 341
ECB,CBC,CFB,OFM, 323
ECB-Mask-ECB Mode, 344
ECDSA, 41, 452
efficiency metrics, 305
electromagnetic emanations, 3
Electronic Code Book Mode, 333
elliptic curve, 157
Elliptic Curves, 172
EM emanations, 407
Embedded Multipliers, 301
Enigma, 508
entropy, 16, 26–29, 31, 38
Entropy Source, 56
entropy source, 18
Epstein TRNG, 60
equivalence relation, 128
Estimation Error, 83
evaluation polynomial, 126
external random number, 27
Extractor Functions, 68

family of distributions, 29–31, 37, 44
fault attack, 47, 48, 50
Fault Attacks, 312
feedback cipher modes, 288
Fermat Number Transform (FNT), 130
Fermat ring, 154
FFT, 416
Fischer-Drutarovský TRNG, 61
folded register, 283
Fourier ring, 129
FPGA, 55, 236, 247, 251, 296
FPGA shift register, 285
functional unit extensions, 204
functional units, 476

Galois field, 239
Gaussian Assumption, 386
GCM, 341
General Extension Fields, 96
Generalized full adder, 109
generalized variance, 34
Golić FIGARO TRNG, 62
Good Curves, 184
Group Law, 176
guessing workload, 26
guesswork, 16

Ha-Moon, 462
Harvesting Technique, 56
hash-and-sign, 367
Hash-ECB-Hash, 341
HCH, 341
HCTR, 341
HECC, 227
HEH, 341
hybrid DRNG, 7, 11, 49
hybrid RNG, 7

hybrid TRNG, 7
hyper-threading attack, 489
Hyperelliptic Curves, 172, 178

IACBC, 341
IAPM, 341
ideal random number, 38
ideal RNG, 5, 39, 41, 44, 45
instruction cache, 476
instruction set architecture (ISA), 191
instruction-level parallelism, 490
instruction-set extension (ISE), 191
integer frame, 137
integer squaring, 80
Intel TRNG, 58
Intentional Current Flows, 382
internal collision, 486
internal collisions, 483
internal random number, 26, 27, 32, 38–40, 42–47
internal random numbers, 14
internal state, 8
Inversion, 119
InvMixColumns, 270, 285
irreducible binary polynomial, 106
irreducible polynomial, 76, 102
irreducible ternary polynomial, 106
irregularities of distribution, 507
Iterative structure, 242
Itoh's Overlapping Windows, 464

Jacobian of a Curve, 173

Karatsuba multiplication, 497
Karatsuba-Ofman, 99, 101
Key Expansion, 247
Key Scheduling, 352
known-answer test, 49
Kohlbrenner-Gaj Design, 63

Lambda Attacks, 181
Leakage Current Flows, 383
Leakage Model, 306
least residue, 136
Left-to-right comb method, 88
LFSR, 28, 32, 33, 37, 40, 43, 49
Liardet-Smart, 457
linear congruential random number generator, 13
linear feedback shift register, 9
Look-Up-Table (LUT), 296
loop unrolling, 254

malicious process, 486
Markov chain, 31, 38, 45, 47
Maurer's test, 28
maximal ideal, 130
Maximum Likelihood, 385
Menezes-Okamoto-Vanstone Attack, 182
Mersenne Number Transform (MNT), 130
Mersenne ring, 154
microarchitectural analysis (MA), 475
microarchitectural attacks, 506
microarchitectural countermeasures, 498
microarchitectural cryptanalysis, 506
min entropy, 16, 28
MIST, 467
MixColumns, 237, 270, 285
mods, 138
modular exponentiation, 125
modular inversion, 125
Modular multiplication, 431
modular multiplication, 125
modular reduction, 81
MonMult, 111
monobit test, 44, 48
MonPro, 436, 445
MonRed, 433
Monte Carlo integration, 13
Montgomery inversion, 121
Montgomery modular reduction, 85
Montgomery multiplication, 367
Montgomery reduction, 431
Montgomery's method, 431
MULGF, 216, 225
MULGF2, 216
multiplicative inversion, 119, 239
Mumford Representation, 175

NIST Test Suites, 57
Noise addition, 310
noise alarm, 42, 45
noise pre-alarm, 45
noise source, 26, 27, 29, 30
Non-adjacent form (NAF), 419
non-feedback cipher modes, 288
non-physical true RNG, 5, 7
NPTRNG, 7, 18, 25, 26
Number Theoretical Transform (NTT), 130
number theory, 507

OCB, 341
OEF, 97
OEF Modular Multiplication, 98
Offset Codebook Mode, 343
one-way property, 19
online test, 26, 27, 39, 42–45, 47–49, 51
OpenSSL, 497

operand scanning, 78
optimal extention field (OEF), 97
optimal normal bases (ONB), 160
Osvik-Shamir-Tromer (OST) Attacks, 487
Oswald-Aigner, 460
outer-round pipelining, 255
Output Feedback Mode, 334
output space, 8
output transition function, 8
overflow, 131

PEP, 341
physical model, 30
physical RNG, 5, 7
physical security issues, 511
pipelined architectures, 258
Pohlig-Hellman Attack, 182
point multiplication, 157
Poisson approximation, 40
poker test, 44
Pollard Rho Attacks, 181
polynomial frame, 128
polynomial modular arithmetic, 76
Post-Processing, 56
Power and EM side channels, 383
practical security, 9
prime field, 110
Principal Subgroup, 174
Processing Unit, 117
processing unit (PU), 112
product scanning, 79
Projective coordinates, 178
Pseudo Number Transform (PNT), 155
pseudorandom number, 27, 28, 42, 45
pseudorandom number generator, 7
pseudorandom numbers, 7
PTRNG, 7, 14, 18, 25, 26, 30, 37, 41–43, 45, 47, 51
PUF-RNG Design, 66
pure DRNG, 7, 8

Quotient Estimation, 82

Rényi entropy, 16, 28
radius, 137, 140
radix, 127
RAM Blocks, 300
random bit generator, 7
random mapping, 39
random number, 5, 25, 27
random numbers, 2
Random pre-charges, 311
random variable, 28
randomized algorithm, 451
randomized exponentiation, 3, 512
raw bit, 18
reconfigurable logic, 55
redundant signed digit (RSD), 108
Reed-Solomon codes, 75
renewal theory, 36
Requirement R2, 2, 6
reseeding, 13
Resilient Functions, 69
Reverse-Engineering, 369
Riemann-Roch, 174
Right-to-left comb method, 88
Rijndael, 237
Rings TRNG Design, 65
RNG, 5, 25
Round transformation, 327
Rounds, 327
RSD arithmetic, 109

S-Box, 274
Schönhage and Strassen, 125
security risk management, 510
seed, 7, 8
seed update, 12
self test, 42
Semaev, Satoh-Araki, Smart Attack, 183
set associative mapping, 479
Shannon entropy, 16, 28, 35
shift-and-xor multiplication, 220
Shift-Register, 300
ShiftRows, 237
Side Channel Analysis, 438
side-channel attack, 47, 50
side-channel attacks, 475
side-channel cryptanalysis, 3
side-channel information, 365
Side-Channel Leakage, 382
side-channel leakage, 3
Side-Channel Resistance, 311
Simple Branch Prediction Analysis (SBPA), 477
Simple Power Analysis, 368
Simple power analysis, 368
Simultaneous Multithreading (SMT), 478, 500
Single bit templates, 393
Slice Multiplexors, 299
slice structure, 299
sliding windows algorithms, 456
smart card implementations, 509
SP-network cipher, 237
Special Modulus, 81, 91
spectral algorithm, 135
spectral coefficient, 129
spectral modular exponentiation, 126

spectral modular multiplication, 126
Spectral Modular Product (SMP), 143
spectral modular reduction, 138
spectral polynomial, 129
spectrum, 129
standard normal distribution, 31
standard random number, 13
state transition function, 8, 11
stationary random variables, 28, 31, 33
statistical blackbox test, 27
stochastic model, 26, 30, 33, 34, 37, 41–43, 45, 47–49, 51
stochastic simulation, 13, 27
SubBytes, 237, 261
subtract with borrow, 78
systolic modular multiplier, 512

T-Box, 276
tamper-resilient cryptography, 511
telescoping sequence, 147
TEMPEST, 407
Template Attacks, 387
ternary extension field, 110
Ternary Extension Fields, 118
test suite, 45, 47
TET, 341
time polynomial, 129
time simulation, 135
time-driven, 480
time-driven attacks, 482
Timing Analysis, 365
Tkacik TRNG, 59
tot test, 42, 46, 48
trace-driven, 480
trace-driven attacks, 482
trace-driven BTB attack, 492
trinomial, 162, 163
TRNG, 7, 12, 15, 25
TRNG designs, 55
true random number generators, 2
true RNG, 5, 7
trusted computing, 506
Trusted Execution Technology (TXT), 476

uncertainty, 2
unified architecture, 113
unified arithmetic, 2, 105
Unintentional Emanations, 410
unpredictability, 2

Virtualization Technology (VT), 476
von Neumann corrector, 68
von Neumann transformation, 38, 41, 46

Weil descent, 183
work factor, 16, 19

XCBC, 341
XECB, 341

Yoo TRNG Design, 67